REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE BADJI MOKHTAR
FACULTE DES SCIENCES DE L'INGÉNIORAT
DEPARTEMENT D'INFORMATIQUE

Année : 2014

Mémoire de fin d'études

Pour l'obtention du diplôme de Magistère
Option : *Informatique Embarquée*

Thème

DEVELOPPEMENT D'UNE CHAINE D'OUTILS EN FONCTION DU NOUVEAU STANDARD FONDATIONNEL UML (FUML)

Présenté par : ZAHOUI Anissa Amel

Directrice de mémoire :

Pr Lynda DIB Professeur à l'Université d'Annaba

DEVANT Le JURY

Président : Pr Salim GHANEMI Professeur à l'Université d'Annaba

Examinateurs : Pr Rachid BOUDOUR Professeur à l'Université d'Annaba

Pr Tahar KIMOUR Professeur à l'Université d'Annaba

Résumé

Ce travail fait partie dans le cadre de L'Ingénierie Dirigée par les Modèles (IDM) et la sémantique d'exécution du langage UML appliqué à l'exécution de modèles des applications embarquées. Dans ce contexte, l'OMG offre une norme qui définit un modèle d'exécution pour un sous-ensemble d'UML appelé fUML (foundational UML subset). Ce modèle d'exécution fournie une sémantique précise et claire qui facilite la transformation de modèles, l'analyse, l'exécution de modèles et la génération de code.

L'objectif de cette thèse est d'étudier et mettre-en-œuvre une chaine d'outils pour les systèmes embarqués, cette chaine d'outils est constitué d'un générateur de code et un simulateur en exploitant les principes portant sur la sémantique d'exécution des modèles à un niveau d'abstraction élevé, dans le but est d'évaluer et étudier la solution dans la phase de modélisation avant de passer a l'étape de réalisation, cela permet à mettre en œuvre une possibilité d'exécution le plus tôt possible dans le flot de conception de l'application. Pour cela, nous avons étendu le modèle d'exécution défini dans fUML en tenant compte sur deux aspects :

- Etude de partitionnement : fUML ne fixe aucun mécanisme d'implantation des tâches : le type : matérielle ou logicielle, et sur quelle ressource va être implémentée afin d'avoir un système partitionné. Pour résoudre ce problème nous introduisons un partitionneur qui définit pour chaque fonctionnalité ; sa ressource de l'architecture, son type d'implémentation.

- Vérification formelle des contraintes attendues : c'est l'évaluation de la solution avant de passer à la phase de réalisation ; pour savoir si le système respecte les contraintes exigées et qu'il répond aux spécifications.

Cette implémentation est réalisée sous forme des plugins Eclipse.

Mots clés : fUML, UML, sémantique d'exécution, modèle, simulation, patitionnement, générateur de code

Abstract

This work is part under the Model Driven Engineering (MDE) and the execution semantics of UML applied to model execution of embedded applications. In this context, the OMG provides a standard that defines an execution model for a subset of UML called fUML (foundational UML subset). This implementation provided a clear and precise semantics that facilitates model transformation model, analysis, model execution and code generation.

The objective of this thesis is to study and develop -implement a tool chain for embedded systems, this tool chain consists of a code generator and a simulator operating principles on the semantics execution model to a high level of abstraction, the goal is to evaluate and study the solution in the modeling phase before moving to the stage of implementation, it allows to implement the possibility of execution as early as possible in the design flow of the application. For this, we extended runtime defined in Fum taking into account two aspects of the model:

- Managed Partitioning fUML sets no mechanism for implementation of tasks: type: hardware or software, and what resources will be implemented in order to have a partitioned system. To solve this problem we introduce a partitioning that defines for each feature; its resource architecture, its implementation type.

- Formal verification of expected constraints: it is the evaluation of the solution before proceeding to the implementation phase; whether the system meets the requirements and constraints that meets specifications.

This implementation is designed as Eclipse plugins.

ملخــــص

هذا العمل هو عبارة عن جزء من الهندسة التطبيقية المخصصة للنماذج (IDM) و مدلول التنفيذ للغة البرمجة (UML) المطبقة على تنفيذ النماذج للتطبيقات المحمولة في هذا المجال. ال OMG وفرت معيار يحدد نموذج تنفيذ لمجموعة فرعية من ال UML تسمى fUML (UML التأسيسية). هذا نموذج التنفيذ يوفر دلالات دقيقة و واضحة في تسهيل تحول النماذج، تحليل، تنفيذ النماذج و إنشاء الشفرات.

الهدف من هذا البحث هو دراسة و وضع مجموعة من الأدوات للأنظمة المحمولة، هذه السلسلة متكونة من منشئ الشفرات و جهاز صوري باستغلال مبادئ، دلالات التنفيذ للنماذج على مستوى عالي من التجريد و الهدف هو تقييم و دراسة الحل في مرحلة نمذجة قبل الانتقال إلى مرحلة التنفيذ في أقرب وقت ممكن في مرحلة تصميم التطبيق لهذا تطرقنا للتعرف على نموذج التنفيذ المعرف في fUML وهذا بأخذ عين الاعتبار النقاط الثلاثة:

- إدارة التقسيم: fUML لا يضع أي آلية لزرع الوظائف، نوع الزرع: عتاد أو برنامج، و على أي وسيلة هندسة ستزرع فيها و هذا للتحصل على نظام مقسم.
- الفحص الشكلي للإشكالات المنتظرة: هي تقويم للحل قبل الذهاب إلى مرحلة الإنشاء لمعرفة إذا كان النظام يحترم القواعد و يتجاوب مع المتطلبات.

تم هذا الزرع على شكل Plugins Eclipse.

L'Ingénierie Dirigée par les Modèles (IDM) propose de modéliser les applications à un haut niveau d'abstraction en utilisant des modèles au cœur des processus de développement des systèmes, puis le code est généré à partie de ces modèles. Un système peut être représenté avec plusieurs modèles selon le niveau d'abstraction spécifique. L'intérêt de l'idm est de diminuer la complexité des systèmes conçus, et garantir la rapidité et la qualité des processus de développement.

Le Model Driven Architecture (MDA) est une discipline offerte par l'Object Management Group (OMG) qui vise à la définition d'un cadre conceptuel (la nature des modèles : indépendants ou spécifiques une plate forme), méthodologique (raffinement des modèles) et technologique (l'utilisation des formalismes : UML pour la modélisation QVT pour les transformations, etc.) pour l'IDM.

Un système embarqué est un système mixte constitué d'une partie matérielle et une partie logicielle, conçu pour exécuter une tâche bien précise, il est intégré dans un environnement. Le partitionnement du système en deux parties logicielle, et matérielle est une étape très importante dans la conception d'un système embarqué.

Le problème de partitionnement se pose lorsque pour une fonctionnalité, nous avons plusieurs composants qui peuvent implémenter cette fonctionnalité, c'est le problème d'allocation des ressources pour savoir chaque fonctionnalité va être réalisée sur quel composant. Mais l'allocation des ressources dépend de l'ordonnancement des fonctionnalités. Pour le système orienté traitement de données, on doit déterminer pour chaque fonctionnalité sa date d'exécution (ordonnancement), et de choisir une implémentation : matérielle ou logicielle (partitionnement). D'où la nécessité d'évaluer la solution avant de passer à la phase de réalisation ; pour que les concepteurs sachent s'ils sont dans la bonne voie, et que le système respecte les contraintes exigées et qu'il répond aux spécifications, donc il est de bonne -qualité.

L'objectif principal de cette thèse est de présenter une chaîne d'outils qui aide à la construction des modèles UML exécutables en fonction du nouveau standard fUML (Foundational UML). Ces modèles exécutables peuvent être construits et testés dans la phase de modélisation. La norme fUML est actuellement en construction et promue par l'OMG pour la construction des modèles UML exécutables.

Nous voulons apporter des modifications sur cette chaine d'outils afin de gérer l'exploration architecturale pendant la phase de partitionnement d'un système embarqué et vérification formelle des propriétés (exigences).

Ce mémoire est organisé en trois parties :

La première décrit le contexte du projet, ainsi que les concepts de l'ingénierie dirigée par les modèles (IDM), le langage de modélisation UML et les systèmes embarqués.

La deuxième partie, présente dans un premier lieu les approches de vérifications précédentes. Nous présentons par la suite en détail le standard fUML de l'OMG.

La troisième partie décrit la contribution de ce travail.

Chapitre 1 Positionnement

1. Contexte du projet

1.1. Faiblesses du langage UML

UML est un langage de modélisation graphique, avec lequel, on peut modéliser la structure statique et la structure dynamique du système, mais UML n'aide pas les concepteurs des systèmes pour les raisons suivantes [1] :

- L'absence d'une précision sémantique formelle, parce qu'UML est un langage semi-formel;
- La complexité d'un modèle en termes de nombre des diagrammes et de symboles, surtout si le système en conception est grand;
- Le non existence de relation ou de dépendance définie entre les digrammes d'un modèle;
- Le manque d'une hiérarchie lors de la décomposition d'un modèle ;
- L'absence de présentation des contraintes et propriétés temps réel dans un modèle ;
- Le manque d'annotations.
- La non existence d'une méthode de transformation à partir du modèle vers l'implantation sans faire assister l'ingénieur.

1.2. La complexité grandissante des systèmes embarqués

Lors du processus de développement, la première difficulté rencontrée par les concepteurs, c'est bien la complexité des systèmes. Il est difficile d'assurer que le système conçu répond aux spécifications souhaitées et qu'il respecte les critères exigés par le client [2].

Cette complexité consiste aussi à fournir des fonctionnalités élaborées, contenant des composants matériels et logiciels, en outre les composants réutilisables permettant pour les concepteurs d'avoir l'avantage de ne pas concevoir à chaque fois les fonctionnalités nécessaires [3]. L'intégration des composants réutilisables est mieux de concevoir à nouveau permet un gain de temps et moins d'efforts par les concepteurs.

De plus, cette complexité a mené les concepteurs à réfléchir, et à réutiliser les systèmes conçus et avoir des méthodes garantissant leur qualité avant la

phase de codage en anticipant la vérification dans les premières phases de conception [4].

2. Ingénierie Dirigée par les Modèles

Dans cette section nous définissons les principes fondamentaux de l'Ingénierie Dirigée par les Modèles notée IDM. Nous décrivons l'approche MDA (Model Driven Architecture) offerte par l'OMG (Object Management Group), qui constitue un espace technologique et un ensemble de standards pour l'application de l'IDM. Dans le but est d'introduire l'approche MDA dans un flot de conception des systèmes embarqués (constitués d'une partie matérielle et d'une partie logicielle).

2.1. Les principes de l'IDM

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) dont le principe est « tout est modèle ». Cette nouvelle approche peut être considérée au même temps comme une continuité et une rupture des travaux précédents en continuité et en rupture avec l'approche objet [5].

Elle est en continuité grâce à la technologie objet qui a mené à l'évolution vers les modèles. En effet, lors de la conception des systèmes sous forme d'objets qui communiquent entre eux, on trouve le problème de classification des objets selon leur origine (objets métiers, techniques,..). L'IDM vise donc, de manière plus radicale à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Le concept fondamental de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Selon les travaux de l'OMG 1, de Bézivin [6] et al et de Seidewitz [7], la définition suivante d'un modèle est considérée [8].

Définition (Modèle) : *Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.*

On déduit de cette définition la première relation capitale de l'IDM, entre le modèle et le système qu'il représente, appelée « représentation de ».

Pour qu’un modèle donne une représentation efficace d’un système qu’il modélise, il doit respecter quelques caractéristiques du modèle parmi lesquels [8]:

- Abstraction : un modèle doit donner une abstraction correcte du système ; c.-à-d. il doit assurer les besoins souhaités de la même façon que le système les assure.
- La compréhensibilité: le modèle ne doit pas être ambigu, facile a comprendre par les utilisateurs. Pour cela il doit être formalisé dans un langage compréhensif.
- La substituabilité: le modèle doit être précis et suffisant c'est-à-dire en substitution avec le système qu’il modélise, le modèle doit contenir les propriétés à étudier par le système.
- L’économie: le modèle doit être peu coûteux par rapport aux coûts du développement du système réel.

Définition *Un méta-modèle est un modèle d’un langage de modélisation. Il définit les concepts ainsi que les relations entre ces concepts nécessaires à la description des modèles.*

La définition de méta-modèle mène à l’identification d’une autre relation entre le modèle et le méta-modèle, appelée « conforme à ».

On dit qu’un modèle est conforme à un méta-modèle si tous les éléments du modèle sont instance du méta-modèle ; et les contraintes exprimées sur le méta-modèle sont respectées [9].

Dans l’IDM il existe deux relations de base :

- La première relation lie le modèle et le système modélisé, elle s’appelle "**Représentation de**".
- La seconde relation lie le modèle et le méta-modèle du langage de modélisation, c’est la relation "**Conforme à**".

La figure suivante (Figure 1) schématise ces deux relations :

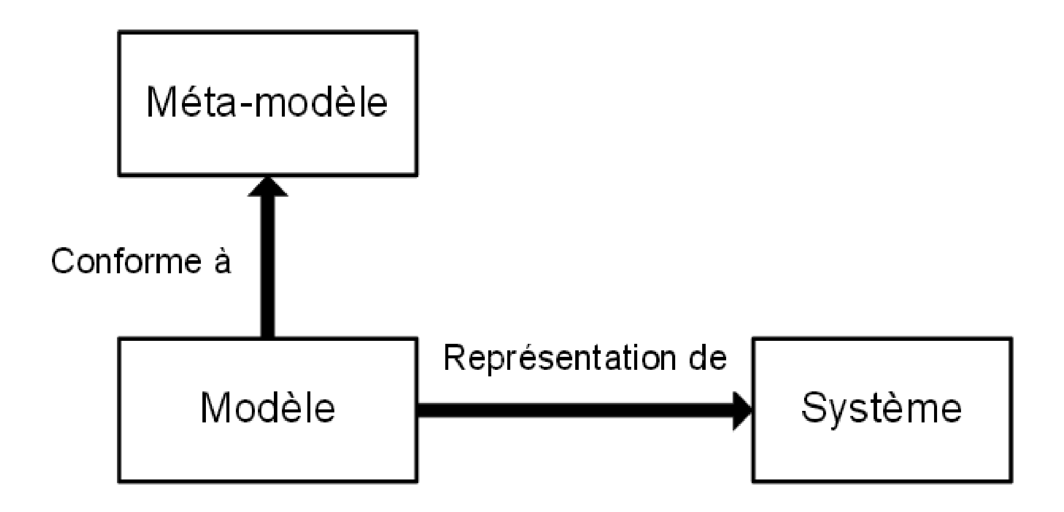


Figure 1: Les relations de base de l'IDM

Un autre concept important dans l'IDM est la transformation de modèles. Elle permet le raffinement des modèles, c'est-à-dire le passage d'un modèle abstrait à un modèle plus détaillé en allant jusqu'à la production des artefacts exécutables [10].

Définition 3 *Une transformation de modèle est une technique qui génère un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources conformément à un ensemble de règles de transformations. Ces règles décrivent essentiellement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.*

La transformation endogène versus transformation exogène : une transformation endogène est une transformation dont les modèles source et cible ont le même méta-modèle tandis qu'ils sont différents pour une transformation exogène [11].

Les transformations de modèle à modèle: selon Tom MENS, ce type de transformation est un raffinement et amélioration de la qualité du modèle. C'est un passage d'un modèle à un autre modèle du même système dont le méta-modèle peut être différent ou le même. Pour cela, ces transformations de modèles nécessitent l'utilisation d'un langage de transformations, Selon Jean BÉZIVIN [12] ces derniers sont classés en ordre chronologique en trois générations :

La première génération concerne les langages de transformation de structures séquentielles d'enregistrement. Dans cette catégorie, un script montre comment le modèle source est transformé en un modèle cible comme les langages de scripts UNIX, AWK ou Perl.

La deuxième génération est les langages de transformation d'arbres. Les modèles sont représentés sous forme d'arbre. La transformation est réalisée par le parcours d'un arbre d'entrée (le modèle source) et la génération d'un arbre de sortie (le modèle cible). Elle se base souvent sur des documents au format XML et l'utilisation de XSLT ou XQuery [13].

La troisième génération forme les langages de transformation de graphes, dans ce cas le modèle à transformer est représenté par un graphe orienté étiqueté. La transformation est donnée cette fois sous forme d'un modèle conforme à un méta-modèle. ATL est un exemple représentatif de langage de transformation de graphes.

Les transformations de modèle à code source: ou génération de code, il s'agit d'un cas particulier de transformation de modèle à modèle où la grammaire du langage de programmation est le méta-modèle du modèle cible. C'est une représentation textuelle des informations du modèle dans un langage de programmation cible [13].

Il existe plusieurs langages de génération de code, par exemple XPand, Acceleo [14].

2.2. L'approche MDA

L'approche MDA (Model Driven Architecture) est un standard proposé par l'OMG dans le domaine de conception des applications afin d'apporter une nouvelle façon de conception et permettre une réutilisation des modèles [15].

Le principe de l'approche MDA est la modélisation de modèle où le modèle métier et est séparé de la plate forme d''exécution autrement dit le modèle est indépendant de la plate-forme technique où le code de l'application est généré automatiquement vers la plateforme choisie.

La technique MDA propose plusieurs outils [16]:

- **UML**,
- **XMI**, XML Metadata Interchange,
- **MOF**, Meta Object Facility,
- **CWM**, base de données pour méta-données.

2.2.1. Les différents modèles du MDA

Le MDA est composé d'un ensemble de modèles, qui ont pour rôle à modéliser l'application, ensuite ces modèles vont subir des transformations successives à fin de générer le code.

Les principaux modèles de la MDA sont [17] :

1. CIM (Computation Independant Model) : modèle indépendant de calcul et de tout système informatique. Il permet de décrire le système mais sans rentrer dans le détail de la structure du système, ni de son implémentation. L'indépendance de ce modèle lui permet de garder tout son intérêt, en plus il est possible de le modifier si les besoins sont changés au cours du temps, qui lui donne une très longue vie.

   Les exigences modélisées dans le CIM seront utilisées dans la construction des modèles PIM (Platform Independent Model) et des PSM (Platform Specific Model).

2. PIM (Plateform Independant Model) modèle qui décrit le système indépendamment de toute plate-forme et de toute technique qui sera utilisé pour le déploiement de l'application. Il décrit le logique métier du système sans détailler son utilisation sur la plateforme; autrement dit il représente le fonctionnement des entités et des services d'où la condition de pérenne 'dure au cours du temps). Ce modèle est décrit en format UML avec la concaténation du langage OCL (Object Constraint Language).

   Nous distinguons plusieurs niveaux de PIM selon le type des informations à représenter : des informations sur la persistance, les transactions, la sécurité,...etc. Ces concepts permettent de transformer plus précisément le modèle PIM vers le modèle PSM.

3. PDM (Plateform Dependant Model) modèle des plates-formes décrit une architecture technique. Il contient des informations pour la transformation des modèles vers une plateforme spécifique. il est considéré comme un modèle de transformation à fin de permettre le passage du PIM vers le PSM.

4. PSM (Plateform Specific Model) est un modèle dépendant de la plate-forme spécifiée par l'architecte. Il sert à générer le code exécutable vers la plate-forme cible en y ajoutant les informations techniques spécifiques à la plate-forme. Il décrit le système d'une façon détaillée du système et sont dépendants d'une plate-forme d'exécution.

2.2.2 La transformation des modèles du MDA

Le principe de développement des applications logiciel dans l'approche MDA est la conception d'un ensemble de modèles et leurs transformations.

Pendant le cycle de développement, il existe plusieurs types de transformations entre les différents types de modèles [9]:

1. Les transformations du type CIM vers CIM, PIM vers PIM et PSM vers PSM servent à améliorer, spécialiser ou filtrer les informations de modèle.
2. La transformation CIM vers PIM permet le passage de la modélisation des exigences vers la description des services sans tenir compte des détails d'implémentation.
3. La transformation du PIM vers PSM permet d'ajouter au PIM des informations spécifiques à la plate-forme d'exécution ciblée en tenant compte les informations du modèle PDM.
4. La transformation du PSM vers du code est la phase de génération du code, après compilation nous obtenons l'exécutable de l'application.

La figure suivante (Figure 2) montre tous les types des transformations décrites ci dessous :

2.3 La Sémantique des langages de modélisation

Un langage de modélisation est nécessaire pour la construction des modèles. Dans le domaine de l'ingénierie dirigée par les modèles, ce langage pourrait être un langage générique ou spécifique. Un langage générique (DGML pour Domain Generic Modeling langage) peut être utilisé pour décrire un grand part des systèmes. Tandis qu'un langage spécifique (DSML pour Domain Specific Modeling langage) inclut des concepts spécifiques permettant de décrire des systèmes ciblés [18].

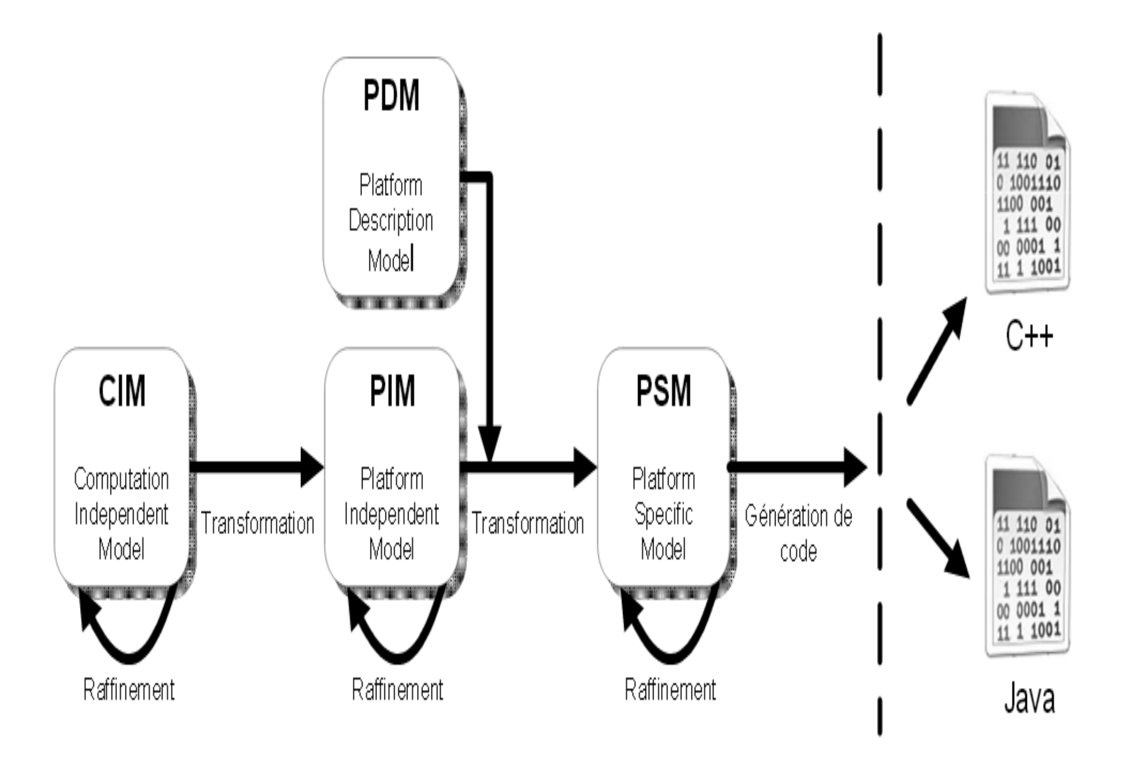


Figure 2: La transformation des modèles du MDA

Dans les deux types de langage de modélisation, la création du langage suit les trois étapes suivantes:

1. Définir une syntaxe abstraite du langage.
2. Définir une syntaxe concrète du langage.
3. Définir une sémantique du langage.

La figure 3 englobe les étapes de création d'un langage de modélisation.

2.3.1. La syntaxe abstraite

La syntaxe abstraite d'un langage de modélisation décrit de manière structurelle, ses concepts et leurs relations. Les langages de méta-modélisation citant le standard offert par l'OMG MOF (Obj 2003a) offrent des concepts et des relations élémentaires sous forme d'un méta modèle pour définir la syntaxe abstraite. A ce jour il existe plusieurs environnement et langages de méta-modélisation: Eclipse-EMF/Ecore(Bu-dinsky et al.,2003), GME/MetaGME(Ledeczi et al.,2001), AMMA/KM3 (ATLAS, 2005) ou XMF-Mosaic/Xcore (Clark et al.,2004) [18] .

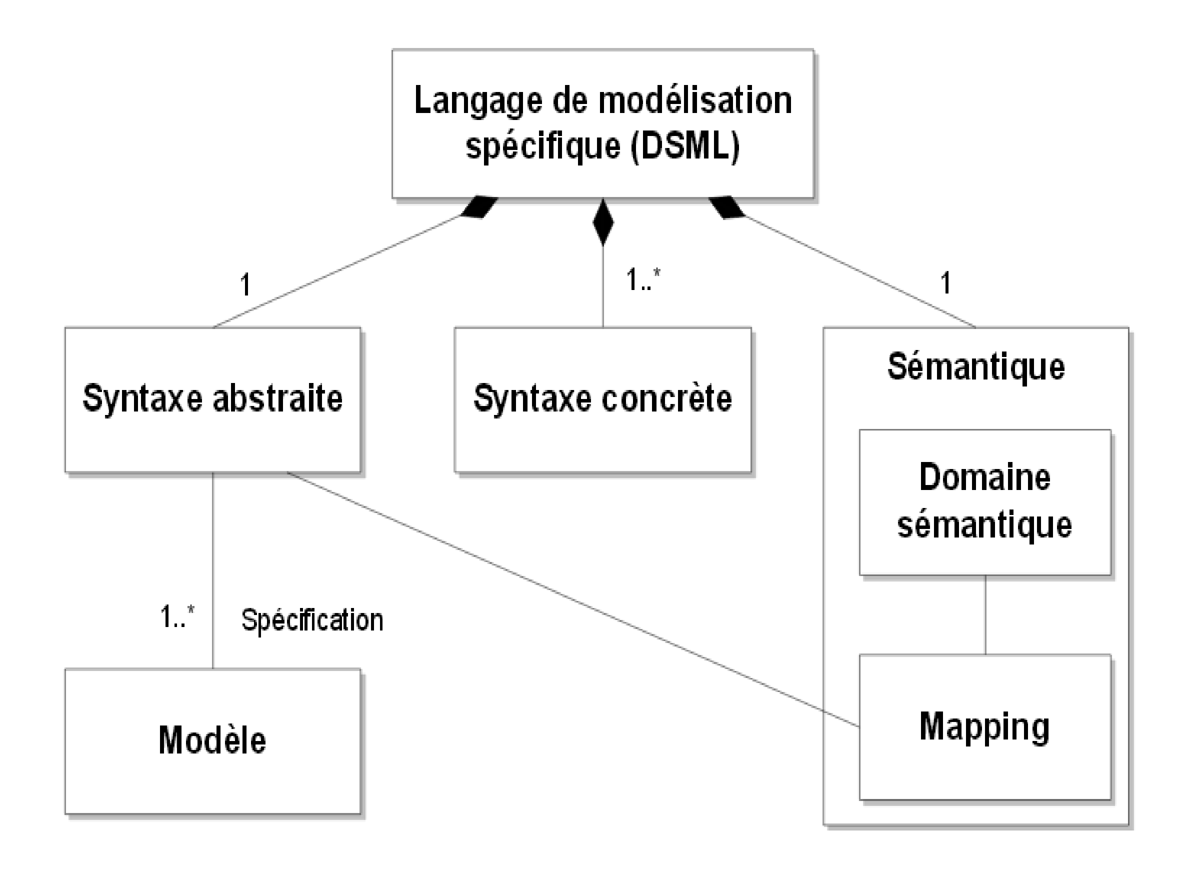


Figure 3: Les étapes de création d'un langage de modélisation

2.3.2. La syntaxe concrète

La syntaxe concrète d'un langage fournit un format qui peut être graphique ou textuel, où les concepts de la syntaxe abstraite sont utilisés afin de générer une syntaxe concrète et obtenir un modèle conforme au méta modèle de la syntaxe abstraite. Pour définir la syntaxe concrète il existe de nombreux outils comme GMF pour des syntaxes graphiques et TCS pour des syntaxes textuelles.

2.3.3 La sémantique

La sémantique d'un langage de modélisation est une description précise et claire des concepts de ce langage. La sémantique est dite formelle lorsqu'elle est formalisée mathématiquement elle décrit précisément et sans ambigüité le sens du modèle [19].

Une sémantique est en fait caractérisée par [20] :

- Un domaine sémantique où les concepts sont clairs et bien définis.
- Un mapping entre les expressions de la syntaxe abstraite et les éléments du domaine sémantique.

Prenons par exemple le langage de programmation C :

- La sémantique d'un programme écrit en langage C est définie en exécutant un ensemble de mot clés qui constituent ce programme, plus précisément, c'est une exécution d'un jeu d'instruction d'un langage cible.
- Le domaine sémantique contient le jeu d'instructions du langage cible.
- Le mapping sémantique est effectué par un compilateur qui transforme le code source (le programme écrit) vers un langage cible constitué d'un jeu d'instructions spécifique.

La sémantique d'un langage de modélisation peut avoir deux types : la sémantique statique ou structurelle et la sémantique dynamique ou comportementale.

La sémantique statique décrit la signification d'un modèle avec des termes et des règles de bonne formation ou les propriétés sont vérifiées avant l'exécution. Par contre la sémantique dynamique décrit le comportement du modèle au cours d'exécution.

D'une façon générale, la sémantique de plusieurs langages de modélisation est décrite de manière informelle, nous citons notamment la norme UML2. Plusieurs problèmes sont soulevés par la sémantique citons par exemple:

- Les normes du langage de modélisation sont ambiguës et mal définies par les utilisateurs. Ces normes des langages doivent avoir une sémantique précise. Un concept d'un langage mal interprété sémantiquement conduit à des mauvaises interprétations et utilisations par les développeurs d'outils, ces conséquences mènent à l'interopérabilité des modèles.
- Une sémantique informelle ne peut être interprétée par les outils, selon leur compréhension de la sémantique, les développeurs implémentent le langage, en conséquence un langage peut avoir plusieurs implémentations. Ainsi, pour la même sémantique, nous pouvons avoir deux outils différents qui ont des implémentations contradictoires.

Pour résoudre les problèmes cités précédemment, chaque langage est lui a été associé une sémantique dans le but est que les modèles possèdent le même sens entre tous les différents acteurs du processus de développement. Cela est spécialement important dans le cas des modèles des systèmes temps réel et embarqués. En effet la sémantique

joue un rôle capital pour déterminer les tendances importantes d'un langage dans une démarche de développement IDM tel que la capacité d'exécution, de transformation ou de vérification.

2.4. Langage de modélisation UML

2.4.1. Historique d'UML

Depuis 1974 jusqu'à 1990, de nombreuses approches objets ont apparues, ensuite en 1994, une cinquantaine de méthodes de conception objets ont existé, parmi elles, trois méthodes sont véritablement émergées:

1) La méthode OMT de James Rumbaugh représente graphiquement les aspects statique, dynamique et fonctionnel du système.
2) La méthode BOOCH'93 de Grady Booch introduit le concept paquetage (package).
3) La méthode OOSE de Ivar Jacobson (Object Oriented Software Engineering) donne une analyse sur la description des besoins des utilisateurs (cas d'utilisation).

Ces trois analystes ont décidé d'unir leur efforts en constituant un langage commun, le fruit de ce travail est proposé en 1996 suite à une requête RFP (Request For Proposal), où la version 0.9 d'**Unifed Medeling Language** (UML) a été présentée.

Le terme *unifed* signifie que les analystes ont établi un couplage des concepts objets.

Le terme *langage* signifie qu'il s'agit de créer un langage de modélisation, et pas une méthode.

En janvier 1997, la version 1.0 d'UML est proposée initialement à l'OMG. Ce dernier ne l'a accepté qu'en Novembre 1997 dans sa version 1.1 où UML a été normalisé par l'OMG comme un standard international.

UML a évalué rapidement. En 1998, UML 1.2 a vu le jour avec des changements cosmétiques.

En Mars 2000, la version 1.3 d'UML a apparue avec des modifications dans le cas d'utilisation et les diagrammes d'activité.

En Septembre 2001, avec l'ajout des nouveaux composants, et profils, la version UML 1.4 est née.

En Avril 2004, une autre version d'UML est créée, c'est la version 1.5.

Ainsi, en Juillet 2005, la version 2.0 est proposée, avec de nouveaux diagrammes et une bonne prise en compte de l'aspect dynamique.

En Avril 2006, la version 2.1est créée.

En Février 2009, UML c'est stabilisé à la version 2.2 [20].

2.4.2. *Définition du langage UML*

UML (Unified Modeling Language) est un langage de modélisation graphique utilisé pour la spécification, la construction, et la documentation des artefacts d'un système. UML est utilisé pour la définition des modèles dans le processus de développement du système, afin de présenter les éléments du système et leur communication.

UML est non seulement un outil graphique intéressant mais une norme qui s'impose en technologie à objets et à laquelle se sont rangés tous les grands acteurs du domaine. Il fournit les éléments permettant de construire le modèle qui sera le langage du projet. C'est un outil de modélisations graphique commun qui permet de représenter et de communiquer les divers aspects d'un système d'information.

Le langage UML est organisé en paquetages, chaque paquetage représente des concepts du système exprimés graphiquement par des diagrammes, chacun s'intéresse à un aspect particulier, ces diagrammes sont classés en deux grandes familles:

A. Diagrammes structurels (Structure diagrams) sont utilisés pour la modélisation structurelle du système. Ils décrivent les entités dans le système:

1. *Le diagramme de classes:* il est constitué des classes, chacune avec sa composition interne (attributs, méthodes,…etc) et chaque classe correspond à un concept du système conçu, des associations, des interfaces et des modules (packages).

Le diagramme de classes couvre l'architecture d'un système, il est utilisé pour la description statique des données et des traitements. Le diagramme d'objet: modélise des exemples de classes. Il décrit le système à un instant particulier à fin de vérifier le diagramme de classes. Le diagramme d'objets utilise des éléments du diagramme de classes.

Illustrant cette description par l'exemple d'un système bancaire représenté par la figure 4. Nous trouvons les classes suivantes : Client, ClientPersonne, ClientEntreprise et Compte. Les flèches à tête creuse

montrent que les classes ClientPersonne et ClientEntreprise héritent de la superclasse Client.

La flèche sous forme d'une ligne relie les classes Client et Compte correspond à l'association « détient » qui joint un ou plusieurs Client à un ou plusieurs Compte.

La classe Compte est répartie en deux parties : la première contient les attributs et la deuxième partie comporte les opérations.

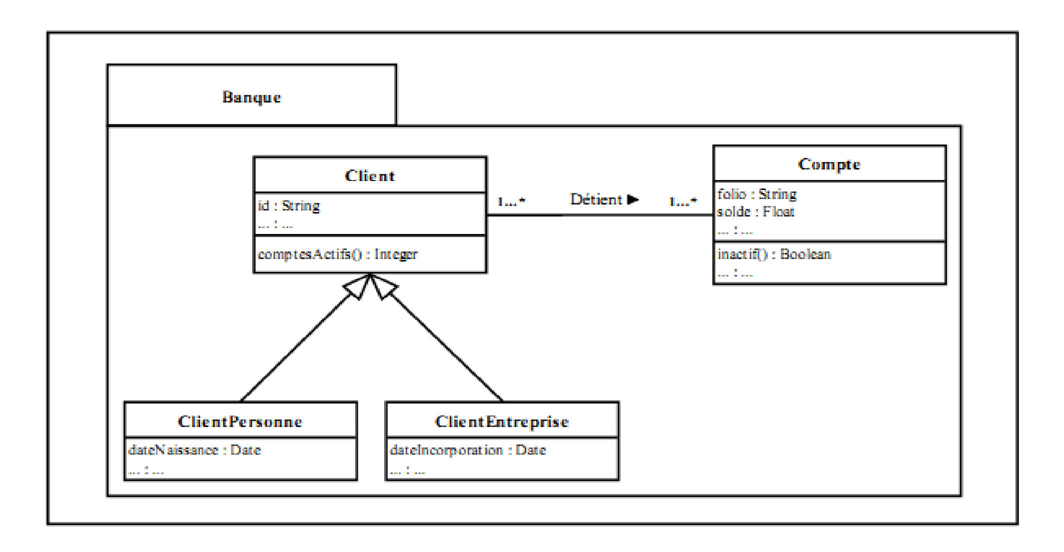


Figure 4:Diagramme de classe d'un système bancaire

2. *Le diagramme de composants:* exprime l'implémentation et le déploiement du système. Il présente une abstraction d'un ensemble de classes qui comprennent différentes fonctions (code source, code binaire, ou code exécutable) et les dépendances entre ces classes issues de l'implantation matérielle et leur interaction avec leur environnement qui est réalisé avec des interfaces.

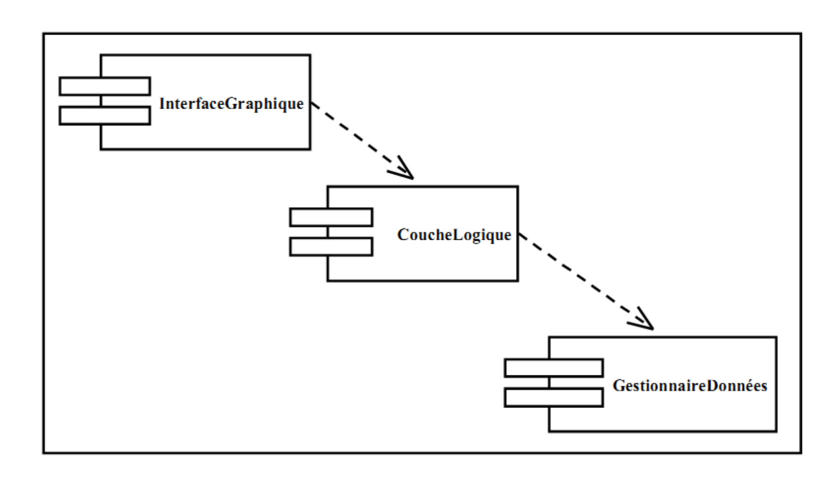


Figure 5: Exemple d'un diagramme de composants

La figure 5 correspond à un diagramme de composants contenant trois composants : InterfaceGraphique, CoucheLogique et GestionnaireDonnées, où le composant InterfaceGraphique dépend du composant CoucheLogique, qui lui-même dépend du composant GestionnaireDonnées.

3. *Le diagramme de déploiement :* il permet d'exposer les composants de l'architecture logicielle et matérielle du système et les liens de communication entre ces composants.

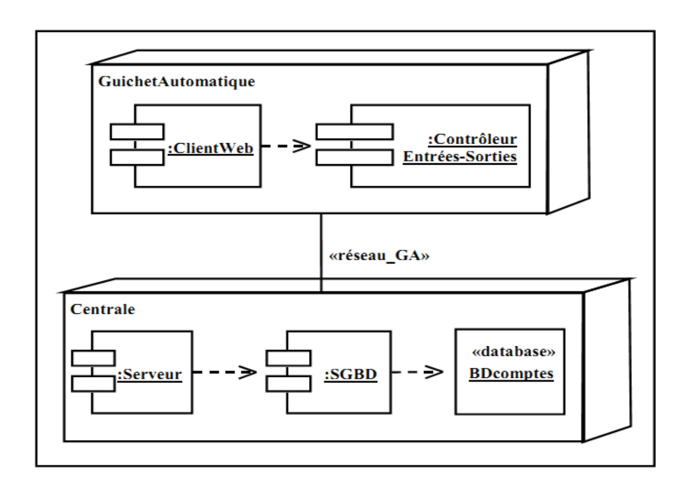


Figure 6: Exemple d'un diagramme de déploiement

La figure 6 illustre une configuration composée de deux unités de traitement : GuichetAutomatique et Centrale qui communiquent à travers un réseau_GA. L'unité de traitement GuichetAutomatique comprend deux composants ClientWeb et ContrôleurEntrées-Sorties. Tandis que le composant Centrale contient deux composantes (Serveur, SGBD) et un objet (BDComptes).

4. Le diagramme de paquetages

5. Le diagramme de structures composites: il décrit la combinaison entre les classes, et leurs composants internes.

6. Le diagramme de profil

B. Diagrammes comportementaux (Behavior diagrams) sont utilisés pour le comportement dynamique du système. Ils présentent ce qui se passe dans le système:

1. *Le diagramme de cas d'utilisation:* ce diagramme est constitué d'acteurs et de cas d'utilisations et il illustre les relations entre ces éléments. Un acteur est une entité extérieure qui déclenche l'un des cas d'utilisation.

Le diagramme de cas d'utilisation englobe les fonctionnalités offertes par le système et décrit son comportement, en identifiant les actions et les interactions entre les acteurs. Ce diagramme est utilisé dans la phase d'analyse à fin de définir les besoins des utilisateurs.

Prenant un exemple présenté dans la figure 7, nous prenons l'exemple d'un système bancaire, nous trouvons deux acteurs : Client et Préposé qui interviennent dans les cas d'utilisation suivants : Ouvrir compte, Effectuer transaction et Fermer compte.

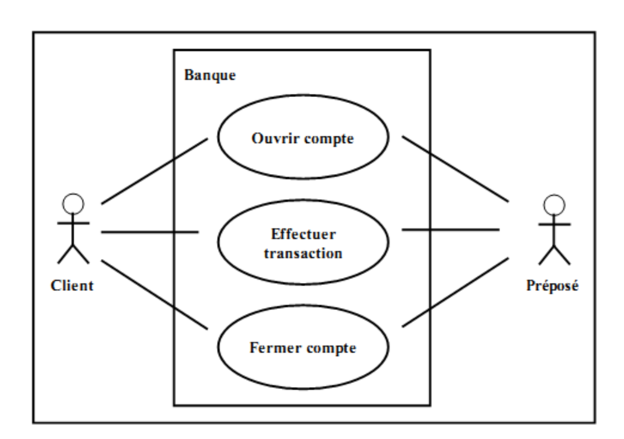


Figure 7: Diagramme de cas d'utilisation d'un système bancaire

2. *Le diagramme d'activités:* il est composé des actions et des transitions entre ces actions. il permet d'exprimer le déroulement et le flot de contrôle interne d'un cas d'utilisation en décrivant le séquencement des activités et leur coordination.

Prenons l'exemple précédent, où nous implémentons le cas d'utilisation Fermer compte. La figure 8 montre le séquencement des activités de ce cas. Chaque activité présente un état. Le flot commence par l'activité du coin supérieur gauche du diagramme d'activité.

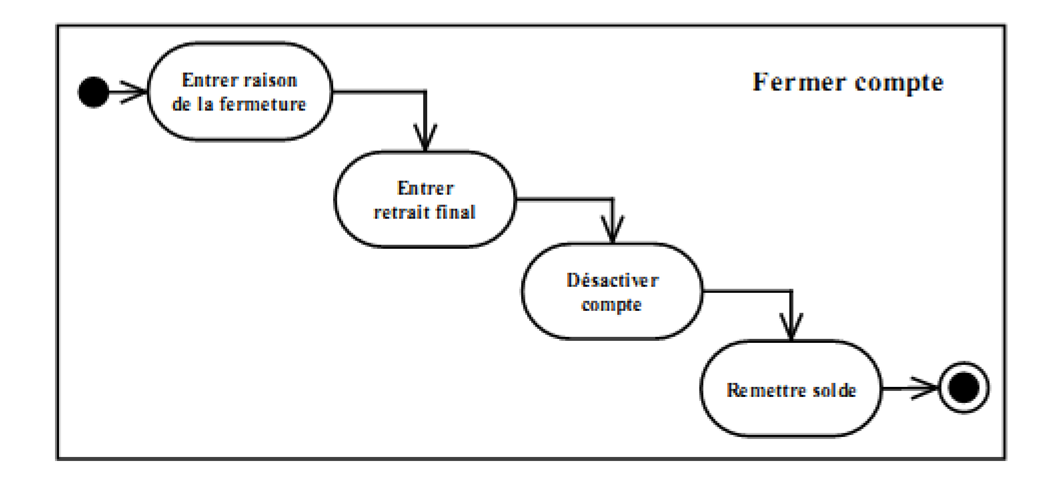


Figure 8: Diagramme d'activité d'un système bancaire pour l'activité Fermer compte

3. *Le diagramme d'états-transitions:* il est composé des états, des transitions et des évènements.

   Ce diagramme exprime le comportement d'une entité du système, il s'intéresse aux évènements externes en couvrant les états possibles d'une classe ou d'un composant. Le changement d'un état d'un objet à un autre état (transition) est provoqué par un évènement responsable à ce changement.

4. *Le diagramme de séquence:* il se compose des objets et des flèches. Ces éléments sont organisés selon deux axes perpendiculaires : le premier axe exprime le temps, et le deuxième axe représente les instances.

   Le diagramme de séquence présente le déroulement d'un cas d'utilisation sous forme d'un scénario entre les acteurs et leurs interactions séquentielles par le changement des messages entre les objets, en respectant l'ordre chronologique (en fonction de temps).

   Dans la figure 9, nous continuons sur l'exemple d'un système bancaire, où il y a quatre objets : GuichetAutomatique, ServeurCentral, ScriptSurServeur et BaseDeDonnées. Entre ces objets, il existe des interactions qui sont représentés par des flèches, où les flèches pleines correspondent à l'invocation des procédures (e.g., solde(), requêteSQL()) et les flèches en traits dénotent leur retour. Nous trouvons aussi chacun des rectangles verticaux correspond à un des deux indications : soit elle présente la période durant laquelle l'objet exécute une action, ou bien que l'objet est en attente d'achèvement d'une sous action imbriquée.

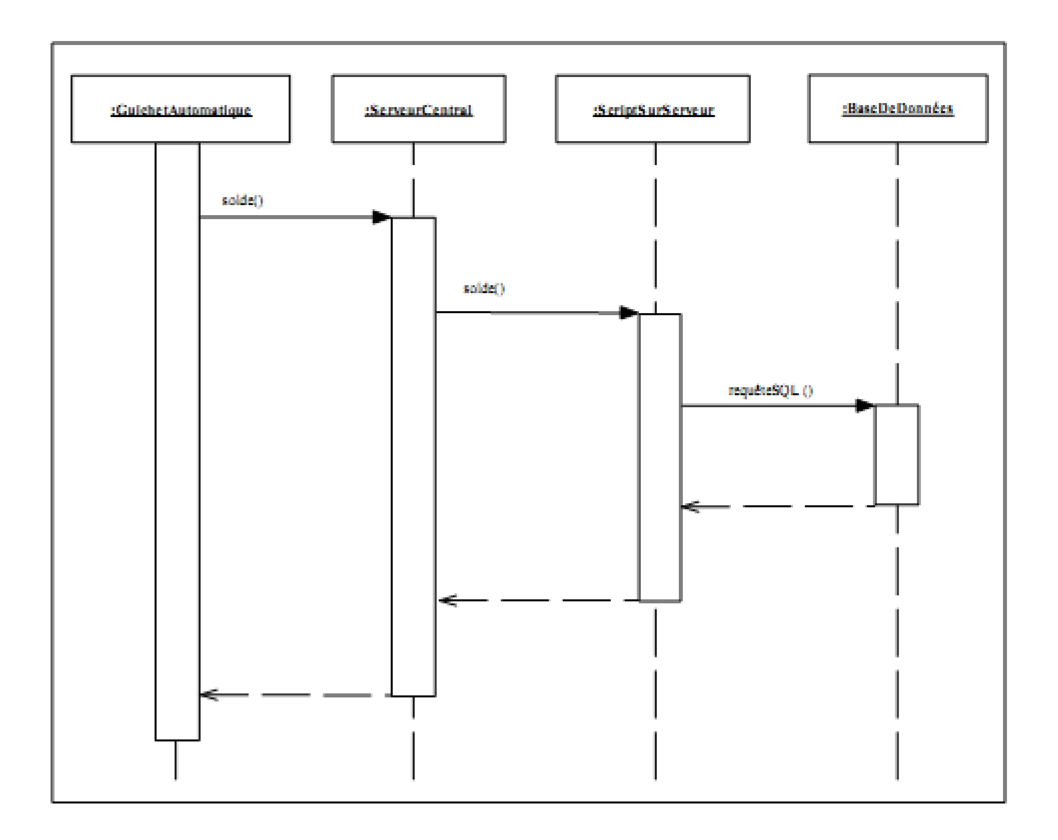


Figure 9: Diagramme de séquence

5. *Le diagramme de collaboration:* il sert à visualiser l'organisation d'un ensemble d'entités qui réalisent un des comportements du système, et il présente le rôle des classes et des associations d'une collaboration, et les interactions entre ces classes.

   La figure 9 présente une collaboration entre quatre classes GuichetAutomatique, FureteurWeb, Terminal et Centrale. Comme il est indiqué dans le diagramme présenté dans la figure 10, les trois premières classes citées précédemment jouent le rôle de Client, alors que la dernière classe joue le rôle de Serveur. Dans les trois associations : GuichetAutomatique, FureteurWeb et Terminal sont associés à des points d'accès provenant du réseau, du Web ou d'une succursale, respectivement; tandis que la classe Centrale représente une connexion selon le type de client.

6. Le diagramme global d'interaction
7. Le diagramme de temps

Ces diagrammes ne sont pas tous utilisés lors d'une modélisation du système, les diagrammes les plus utilisés sont les diagrammes d'activités, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions.

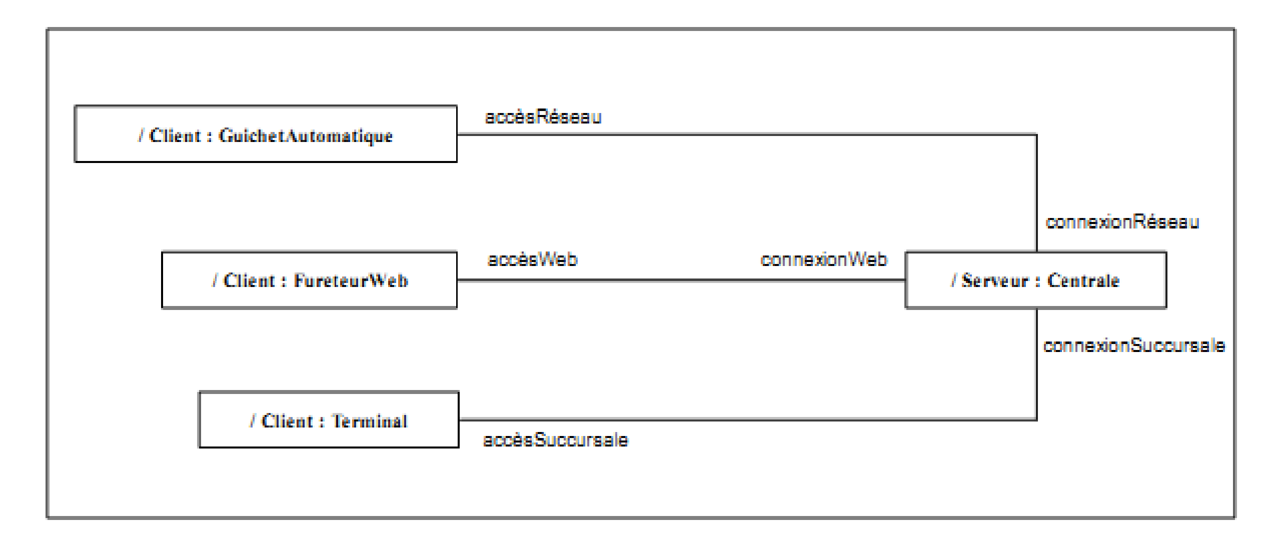


Figure 10: Diagramme de collaboration d'un système bancaire

3. Les systèmes embarqués

3.1. Définition

Un système embarqué est un système électronique et informatique autonome, qui ne possède pas d'entrées/sorties standards (clavier, souris,…etc). IL est généralement produit pour résoudre un problème bien précis, pour exécuter une tâche bien définie [21].

Un système embarqué est considéré comme un système mixte constitué d'une partie matérielle et une partie logicielle, il interagit avec l'environnement lequel il appartient [22].

3.2. Architecture d'un système embarqué

Le système est composé de plusieurs couches. La couche la plus abstraite est le logiciel spécifique de l'application. Cette couche communique avec une deuxième couche s'appelant le système d'exploitation qui contient deux parties : la première partie de la gestion des ressources et la deuxième partie de communication logicielle, suivie par la couche de réseau de communication matérielle embarquée, et à la fin on trouve la couche des composants matériels qui est la plus basse [23].

- *La couche basse:* comprend les composants matériels du système, parmi ces composants, on trouve les composants standards : (DSP, MCU,…etc).
- *La couche de réseau de communication matérielle embarquée:* contient les dispositifs nécessaires à la communication entre les composants.

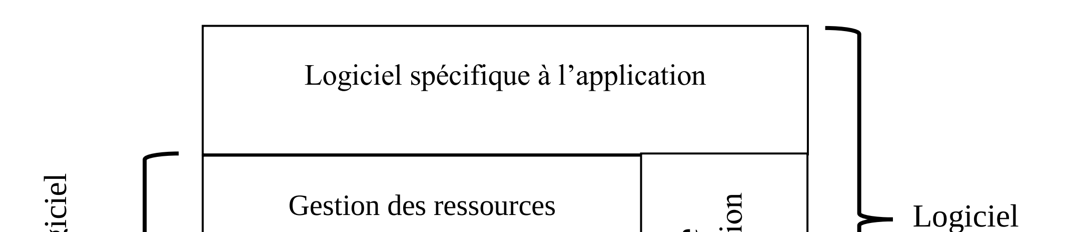


Figure 11: représentation des différentes couches d'un système embarqué

- *<u>Le système d'exploitation comprend deux couches:</u>*

1) <u>Une couche de communication logicielle:</u> composée des pilotes d'entrés/sorties, des interruptions, ainsi que les contrôleurs qui sont utilisées pour contrôler le matériel. Le logiciel de cette couche et le matériel sont intimement liés. Le logiciel et le matériel sont séparés grâce à cette couche.

2) <u>Une couche de gestion des ressources:</u> qui utilise des structures de données particulières et des accès non standards que l'on ne trouve pas des OS standards, par contre, les couches de gestion des ressources des OS standards sont souvent volumineuses afin d'être embarquées. La couche de gestion des ressources permet d'isoler l'application de l'architecture.

- *<u>La couche de logiciel spécifique de l'application:</u>* est la couche la plus abstraite, elle communique avec la couche de plus bas niveau qui est la couche système d'exploitation.

3.3 La structure de base d'un système embarqué

Le système embarqué interagit avec son environnement pour lequel il rend des services bien précis (contrôle, surveillance, communication, …). Une information est captée par l'environnement, une transformation est réalisée sur cette information, avant d'être lisible par le cœur du système embarqué (constitué d'une partie matérielle et une partie logicielle), qui effectue un traitement spécifique à cette information pour rendre à son tour une réponse à son environnement, cette réponse doit être transformée avant d'être envoyée à l'environnement [24].

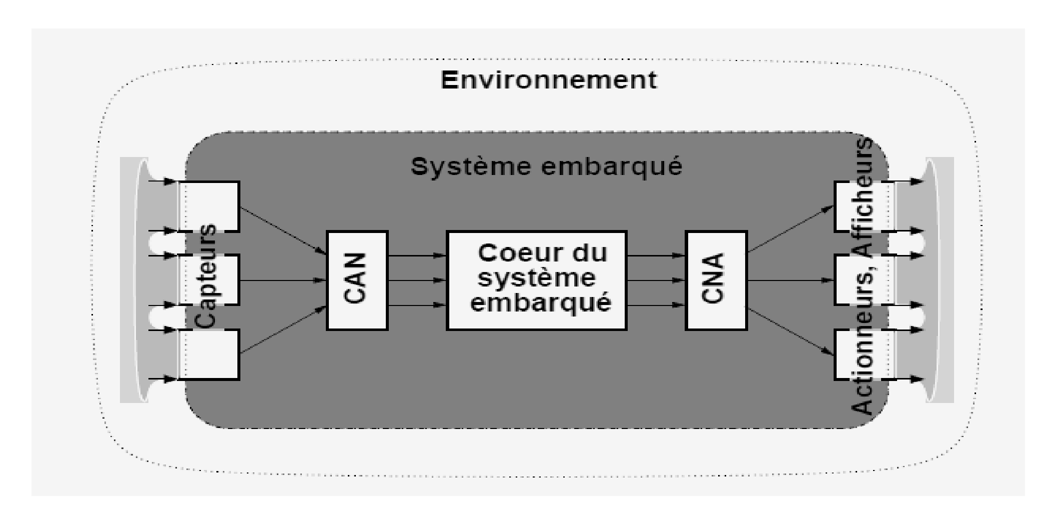


Figure 12: Structure de base d'un système embarqué dans son environnement [25]

3.4 Contraintes d'un système embarqué

- L'encombrement: les systèmes embarqués doivent être de faible poids parce qu'ils sont souvent portables.
- Consommation : la consommation d'énergie doit être minimisée parce qu'ils sont transportés.
- Adaptation à l'environnement : parce que les systèmes embarqués n'évoluent pas dans un environnement contrôlé, d'où l'importance de l'évolution des caractéristiques des composants selon l'environnement afin d'adapter avec; citant : le changement de la température, les interférences RF, la corrosion, les radiations, les vibrations, …etc).
- Fonctionnement en temps réel: les systèmes embarqués doivent agir à temps, parce que la validité du résultat est estimée à son temps de délivraison, et le non respect du délai peut causer des erreurs de fonctionnement, même des dégâts.
- Coût : les systèmes embarqués doivent être de faible coût.
- Sureté de fonctionnement: les systèmes embarqués doit toujours fonctionner correctement même dans le cas de la panne d'un composant.
- Sécurité : à cause de leur utilisation dans le domaine de l'aéronautique, les systèmes embarqués doivent être sécurisés.
- Temps de mise sur le marché: il faut fabriquer des systèmes plus rapides pour encaisser le marché [25].

3.5 Le codesign

a) *<u>Définition</u>*

Le codesign se traduit par la conception conjointe du matériel et du logiciel, c'est le processus de conception, ou à partir des spécifications de cahier des charges, on conçoit un produit constitué d'une partie matérielle et une partie logicielle, avec satisfaction des contraintes exigées [26].

Le codesign intègre donc dans un même environnement la conception du matériel et du logiciel.

Donc l'objectif souhaité via le processus de conception est d'avoir un meilleur compromis matériel/logiciel.

b) *Les phases de codesign*

- *La première phase: spécification :* Les exigences décrits dans le cahier de charges sont reformulés en détail et d'une façon exhaustive afin de définir les spécifications du système sans se préoccuper du découpage de l'architecture en une partie logicielle et une partie matérielle. Le résultat de cette phase est que le concepteur spécifie ce que veut l'utilisateur, et le client est satisfait que ses besoins et ses exigences soient définis.

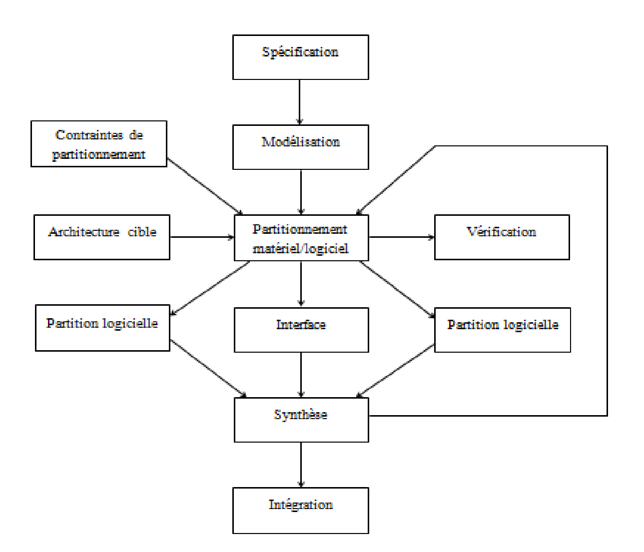


Figure 13: Les étapes de codesign pour le développement d'un système embarqué [27]

- *La deuxième phase: modélisation :* c'est la phase dans laquelle les spécifications sont représentées sous forme d'un modèle. Le modèle est une représentation simplifiée du système en tenant compte de ses propriétés pertinentes et son comportement temporel et fonctionnel sans considérer sa réalisation. Il existe plusieurs formalismes de modélisation, on peut citer [28]:

1) Automates à états finis (FSM: Finite State Machines);
2) Diagrammes flots de données (DFD: Data Flow Diagrams);

3) Réseaux de Petri RDP;

4) UML.

- *La troisième phase: partitionnement:* Le rôle de cette étape est la création d'une architecture composée d'une partie matérielle et d'une partie logicielle à partir des spécifications et de trouver le meilleur compromis entre les parties logicielles et matérielles en fonction de leurs interactions et des contraintes imposées notamment par des critères de performances et de réutilisation. D'une façon générale une réalisation logicielle est effectuée afin de réduire le coût de conception, et une réalisation matérielle est choisie pour augmenter la performance. Le partitionnement englobe à trois points essentiels [29]:

1) **Le choix de la plate forme d'exécution:** Ce choix est basé sur ce que existe, on peut être en trois cas :

   a) *La réutilisation de l'existant :* où la plate forme d'exécution existe et elle est réutilisable, les briques de bases existent, il s'agit de logiciel de bas niveau et les drivers, il reste que l'ajout de l'applicatif.

   b) *Adaptation de la plate forme d'exécution:* dans ce cas, il est nécessaire de concevoir le cœur du système et les périphériques supplémentaires.

   c) Conception et réalisation d'une nouvelle plate forme d'exécution: c'est le cas le plus délicat, où le concepteur de réaliser une nouvelle plate forme d'exécution.

2) **L'allocation des ressources:** c'est le placement des fonctionnalités, c'est-à-dire, chaque fonctionnalité du système est affectée à un composant matériel ou logiciel qui convient s'il existe selon les caractéristiques de ce dernier (composant), sinon on considère les contraintes exigées sur cette fonctionnalité.

3) **L'ordonnancement des fonctionnalités:** consiste à choisir un ordre dans lequel les fonctionnalités seront exécutées pour répondre à certains critères citant : le temps, la performance, en prenant en considération les dépendances entre les fonctionnalités.

- *La quatrième phase: vérification:* dans cette phase une simulation des parties matérielle et logicielle est effectuée dans le but de :

- Vérifier le respect des contraintes dynamiques du système produit ;
- Vérifier si le système conçu répond aux exigences (spécifications) fonctionnelles et non fonctionnelles, et il respecte toutes les contraintes exigées.

La vérification se fait à l'aide des outils formels.

- ***La cinquième phase: synthèse:*** c'est l'étape de réalisation des composants matériels et logiciels

3.6 Ingénierie dirigée par les modèles pour les systèmes temps réel embarqués

Dans le but est de concevoir des applications temps réel embarqués sûres, fiables et efficaces, des spécifications précises, complètes et non ambiguës sont nécessaires pendant le processus de développement.

L'ingénierie dirigée par les modèles apporte de bonnes pratiques pour le développement de logiciels. Ces pratiques sont ajoutées lors du développement des applications temps réel embarquées pour de nombreuses raisons [30]:

- La spécification des applications temps réel embarquées peut avoir des points de vue différents (ex. fonctionnel, temps-réel, tolérance aux fautes, etc.) D'où la nécessité d'ajouter qui des techniques d'abstraction lors du développement.
- Les options d'implémentation peuvent changer considérablement ; où pour un même modèle, nous pouvons voir plusieurs différents modèles d'exécution selon les contraintes de réalisation particulières (modèle multi-tâches, communication synchrone, programmation en boucle, etc).
- La contrainte de performance des applications temps réel embarquées diminue lors de l'utilisation des techniques standards pendant le développement du logiciel. Les optimisations réalisées engendrent un code fonctionnel qui provoque la non fidélité du système final.
- Le test et la validation des applications temps réel embarquées sont des contraintes cruciales, en effet la définition de modèles et d'outils d'analyse précis et spécifiques est nécessaire.

En effet, lors de l'utilisation de l'IDM lors du développement des applications temps réel embarquées, elle mène à la définir un ensemble d'artéfacts (règles méthodologiques, transformations de modèles, génération automatique de code), qui doit être homogène, intact et auparavant testés, assurés et évalués. En outre l'ingénierie dirigée par les modèles introduit un inconvénient dont le code généré est moins performant qu'un code optimisé directement pour une plateforme cible.

Chapitre 2 Etat de l'art

Lorsqu'on cherche une définition de la vérification en informatique, il suffit d'ouvrir le dictionnaire: Vérification n.f. action de vérifier, de s'assurer de l'exactitude de quelque chose en le confrontant avec ce qui peut servir de preuve. Encore la vérification est une action de contrôler quelque chose pour s'assurer de sa conformité, de sa légalité. Cette définition laisse cependant un doute sur ce que l'on vérifie exactement. En effet, le but de la vérification n'est pas de créer des logiciels sans erreurs. Ce but est extrêmement difficile à éteindre, voire impossible.

Le but de la vérification informatique est plus fonctionnel. Il s'agit de prouver qu'un ensemble de contraintes (propriétés), que les concepteurs veulent atteindre, est vrai sur le logiciel. Pour ce faire, il existe plusieurs méthodes de vérification (analyse statique, preuves automatiques et vérification de modèle,…etc).

1. Les techniques de vérifications

1.1. L’analyse statique

L’analyse statique correspond à l’ensemble des techniques qui permettent de déduire algorithmiquement des propriétés sur le comportement d’un logiciel à partir de l’analyse de son code source et/ou de son code assembleur.

Typiquement, l’analyse statique est utilisée lors de la compilation dans le but de détecter des bogues usuels et optimiser le code assembleur obtenu, sans modifier le comportement du programme. Le compilateur va construire un arbre de syntaxe abstraite à partir du code source qui représente toutes les exécutions possibles. Par la suite cette structure est convertie en code assembleur. Le compilateur va vérifier les propriétés sur l’arbre construit.

Ce dernier va modifier afin d’avoir un code assembleur optimisé avec une condition qu’aucune de ces modifications modifie le comportement du programme du point de vue de l’utilisateur (le programme d’origine et le programme optimise doivent rester bisimilaires).

L’analyse statique permet de vérifier des propriétés telles que :

- Une variable est réutilisée par la suite ?
- Est-ce que il y a un accès à l’élément p+1 dans un tableau qui contient p éléments ?
- Le contenu d’une variable peut être corrompu par des accès aléatoires à la mémoire ?

Néanmoins, cette technique a ses limites. Comme son nom l’indique, il s’agit d’analyse statique et non dynamique. C’est à dire que l’on ne peut vérifier que les variables qui sont initialisées comme des variables statiques dans le programme. Ce qui élimine d’emblées toutes les variables dont l’initialisation est dynamique.

Pourquoi ne serait-il pas possible d’appliquer la même analyse aux variables dynamiques ?

Tout simplement parce que l’analyse statique revient à explorer toutes les exécutions possibles et si certaines variables sont dynamiques, le nombre des exécutions possibles est infini. Mais, peut-être existe-t-il un algorithme qui réduise le nombre de ces exécutions infini à un nombre fini ? En fait, il a été prouvé que cela n’est pas possible ; c’est ce qu’on appelle le théorème de Rice : Toute propriété extensionnelle non triviale de programmes écrits dans un langage récursivement énumérable est indécidable.

A l’heure actuelle il existe des résultats qui ordonnent l’analyse statique et l’interprétation abstraite afin d’inclure une partie de ces variables dynamiques dans la vérification [31].

La formalisation dans un langage de type langage de description de matériel n'est pas suffisante pour une approche de *preuve formelle*. Cette approche, alternative à la simulation, offre des techniques de validation parfaitement fiables, car il s’agit d'utiliser des méthodes "mathématiques" pour vérifier formellement l'adéquation du système à son comportement espéré (ou bien détecter d'éventuelles erreurs). Sans tester le comportement du système sur des jeux d'essais aussi significatifs que possible. Ce raisonnement formel nécessite la transformation de la description VHDL, Verilog,... dans une forme adaptée aux outils qui vont être mis en œuvre pour la preuve formelle.

1.2. Vérification formelle

Les vérifications basées sur les preuves de théorèmes

1.2.1. Vérification de modèles (Model-cheking)

Le modèle-checking consiste à vérifier si un modèle d’un système ou une abstraction, satisfait une propriétéà l’aide d’un traitement de toutes les exécutions possibles en parcourant d’une manière exhaustive le graphe d’état correspondant au système.

Le grand avantage de cette technique est dans le cas où aucune exécution satisfait la spécification vérifiée, elle fournit un contre-exemple qui permet d’analyser le bug éventuel d’où une correction plus rapide.

L’analyse de toutes les exécutions possibles nécessite de contrôler chaque état par rapport à la propriété avec la possibilité de retour sur des étatsprécédents pour lesquels il existe des transitions activées mais non explorées.

La vérification de modèles correspond à l’ensemble des techniques qui permettent de déduire algorithmiquement des propriétés à vérifier sur le comportement d’un système (logiciel, algorithme ou protocole) en utilisant des notations prédéfinis comme un modèle (automate fini, automate temporise, réseau de Pétri, algèbre de processus, ...).

La première étape consiste à produire un modèle du logiciel, algorithme ou protocole que l’on veut vérifier. Le formalisme de ce modèle est un système de transition étiquetée. Il convient ensuite de traduire les propriétés que l’on veut vérifier en formules logiques (LTL,CTL, CTL*, TCTL, FOL, ...).

La dernière étape, c'est la vérification proprement dite, elle est totalement la tâche effectuée par la technique de vérification. Celle-ci va ranger le modèle et la formule logique et calculer l'ensemble des états accessibles en avant (post ou forwardanalysis') ou en arrière (pre*ou 'backwardanalysis'). S'il existe un chemin entre l'état initial et l'ensemble des états qui vérifient la formule, alors la propriété est vérifiée.

Il existe plusieurs techniques permettant de modéliser et de vérifier des propriétés par ce biais :

1.2.1.1. Les réseaux de Pétri

Les réseaux de Pétri [32] sont utilisés afin de modéliser le comportement dynamique de systèmes discrets. Ils sont composes de deux types d'objets : les places et les transitions. L'ensemble des places permet de représenter l'état du système ; l'ensemble des transitions représente alors l'ensemble des événements dont l'occurrence provoque la modification de l'état du système.

A l'occurrence d'un événement correspond le franchissement d'une transition dépendant de la satisfaction de préconditions.

Les réseaux de pétri offrent une représentation graphique simple des systèmes modélisés. Une place est représentée par un cercle, une transition par un rectangle, et les relations de causalité au sein du système sont représentées par la présence - ou l'absence - d'arcs valus reliantles places aux transitions.

Un des atouts, et non le moindre, de ce formalisme est d'offrir de nombreux outils d'analyse structurelle, tels que la théorie des réductions ou le calcul d'invariants.

Un certain nombre d'outils utilisent cette théorie comme base, on peut citer : DESIGN-CPN [33], PAPE-TRI [34] et PEP [35].

1.2.1.2. Les automates à pile

Les automates à pile sont des automates dont l'ensemble des états accessibles est reconnaissable et calculable et dont le model-checking des formules CTL est décidable.

1.2.1.3. Les automates temporisées

Les automates temporisées [36] dont le vide est décidable ainsi que le model-checking de TCTL et d'un certain nombre d'autres

logiques temporelles temporisées. Plusieurs outils utilisent cette théorie Kronos [37], Uppaal [38] et CMC [39].

1.2.1.4. Les automates hybrides

Les automates hybrides sont un cas à part. L'accessibilité est indécidable mais certains semi-algorithmes efficaces dans une grande partie des cas réels existent. L'outil le plus connu dans ce domaine étant HyTech [40]. La vérification de modèles pose deux problèmes cruciaux.
Le premier apparaît lors de la conception du modèle. En effet, il est crucial de conserver une équivalence entre le modèle que l'on va utiliser et la réalité (au moins en ce qui concerne les propriétés que l'on vérifie).

De cette équivalence dépend la validité de la preuve que l'on apporte. L'autre problème vient de l'explosion du nombre d'états du modèle que l'on vérifie. Ce problème se rencontre quasiment systématiquement sur les algorithmes de calcul de pré ou de post. C'est bien souvent ce problème qui empêche la vérification de systèmes trop complexes. Il est cependant possible de réduire le nombre des états par le choix d'abstractions qui préservent les propriétés voulues, tout en réduisant le nombre d'états à parcourir.

1.3. Simulation

Cette technique est utilisée dans n'importe quelle phase de conception avec la présence d'un modèle de calcul.

Le rôle des *simulateurs* est d'étudier le comportement dans un temps du système défini, en utilisant des batteries de jeux d'essais ; c'est une méthode qui permet de valider le système avant son construction, la simulation est une méthode fiable dans le bas niveau d'abstraction et les jeux d'essais peuvent garantir quelques cas, ce qui conclue qu'elle est moins rapide la vitesse et moins fiable [41].

2. fUML

2.1. Définition

Du au problème d'ambigüité de la sémantique d'exécution des éléments d'UML (la sémantique est exprimée en langage naturel), d'où la nécessité de décrire les modèles dans un langage ayant une sémantique d'exécution bien définie, c'est le modèle UML exécutable.

Le sous-ensemble Fondationnel UML ou fUML. Il s'agit d'un langage complet pour les modèles exécutables [42].

Un objectif fondamental de fUML est de servir d'intermédiaire entre le «sous-ensembles» de la modélisation UML et les langages plate-forme utilisés comme cible pour l'exécution du modèle. Comme le montre la figure 14.

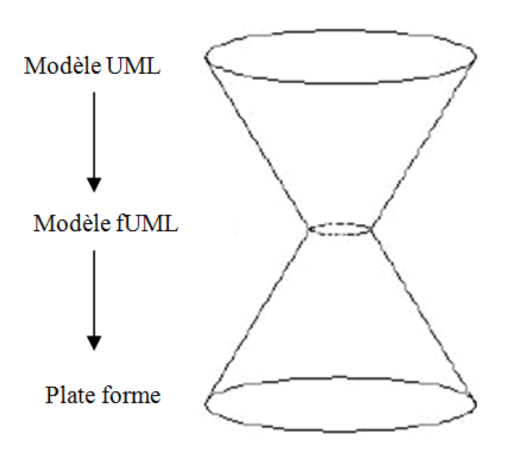


Figure 14: Transformation de et à le sous-ensemble foundational UML

Avec l'adoption du standard fUML, l'OMG a défini une sémantique d'exécution claire, précise, formelle et non ambigüe pour un sous ensemble d'UML. Ce sous ensemble considère quelques concepts d'UML.

La sémantique d'exécution de fUML est décrite sous forme d'un modèle d'exécution. Le modèle d'exécution englobe des concepts : les classes et les associations qui couvrent l'aspect structurel de modélisation, et pour l'aspect comportemental, on trouve les activités et les actions. Ces packages retenus par le fUML sont présentés dans le tableau suivant :

Les packages UML	Inclus dans fUML ?
Modélisation structurelle	
Classes	Oui
Composants	Non
Structures composites	Non
Déploiement	Non
Modélisation comportementale	
Actions	Oui
Activités	Oui
Comportements	Oui
Interactions	Non
Machines à états	Non
Cas d'utilisation	Non

Tableau 1: Les packages retenus dans fUML

2.2. La syntaxe abstraite de fUML

Les classes

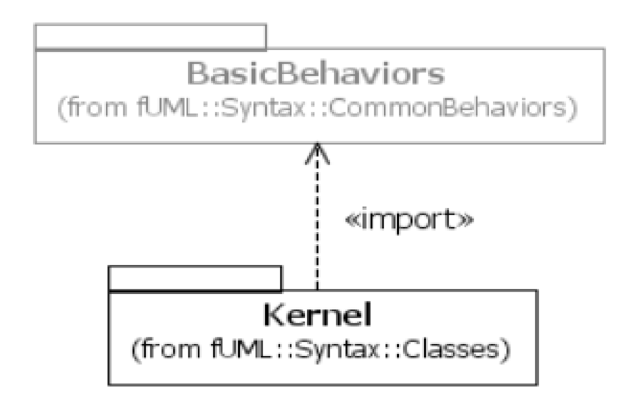


Figure 15: La syntaxe du package de la classe

CommonBehaviors

Le package CommonBehaviors comprend deux sous-ensembles :

- BasicBehaviors
- Communications

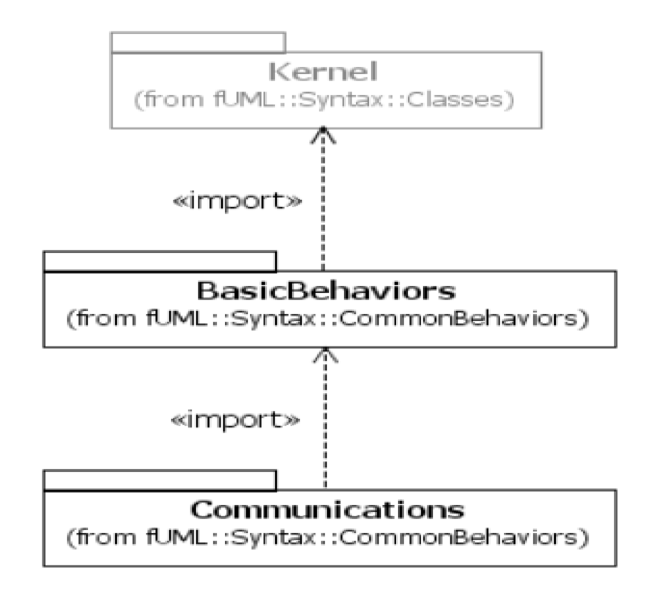


Figure 16: La syntaxe du packageCommonBehaviors

Les activités

Le package des activités inclut 3 sous-packages :

- IntermediateActivities
- CompleteStructuredActivities
- ExtraStructuredActivities

La figure 17 montre la dépendance entre ces packages.

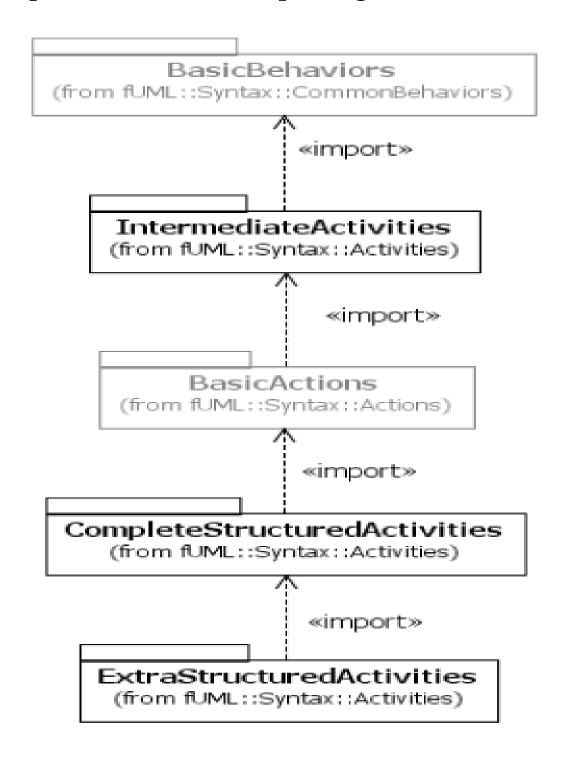


Figure 17: La syntaxe du package activités

<u>Les actions</u>

Ce package inclut trois sous packages :

- BasicActions
- IntermediateActions
- CompleteActions

La figure 18 montre la dépendance entre ces packages.

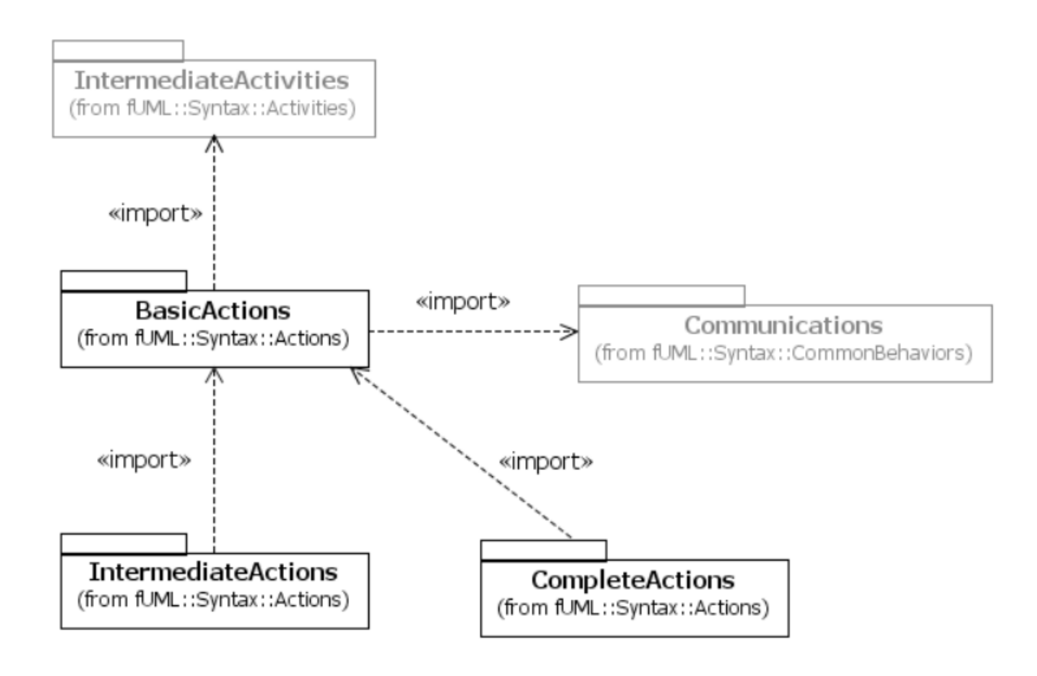


Figure 18: La syntaxe du package actions

2.3. La sémantique : Le modèle d'exécution de fUML

Le modèle d'exécution est une spécification opérationnelle en Java de la sémantique de fUML. Il spécifie généralement la sémantique d'exécution des concepts de modélisation compromis dans le sous-ensemble fUML.

Le modèle d'exécution est lui-même un modèle défini par un sous-ensemble appelé bUML (base UML). Cette définition circulaire est rompue par une description axiomatique de premier ordre de la sémantique.

La structure du modèle d'exécution est spécifié en packages, conformes au packages de la structure du modèle de la partie syntaxique. Chaque package de la syntaxe abstraite a un package correspondant dans le modèle d'exécution sauf le package nommé Loci, qui n'a pas de package symétrique avec les concepts syntaxiques. Ce package jour le rôle d'un moteur et l'environnement d'exécution des modèles fUML.

La définition du modèle d'exécution de fUML est basée sur la classe principale de conception qui s'appelle **visiteur**. Son rôle est de raccorder à une hiérarchie de classes existante sans modifier la structure. Dans le contexte de fUML, il permet d'attacher un comportement aux concepts de modélisation sans changer le méta-modèle de la syntaxe abstraite [43]. Plus précisément, pour chaque méta-classe de fUML, un comportement est donc raccordé via une classe visiteur dans le modèle d'exécution.

Chaque nouvelle classe a une association unidirectionnelle avec la méta-classe de l'élément syntaxique et définit les opérations nécessaires pour capturer la sémantique de cet élément.

Toutes les classes de type **visiteur** héritent directement de la classe **SemanticVisitor** du modèle d'exécution. Il existe trois types de classes visiteurs, présentées dans la figure 19.

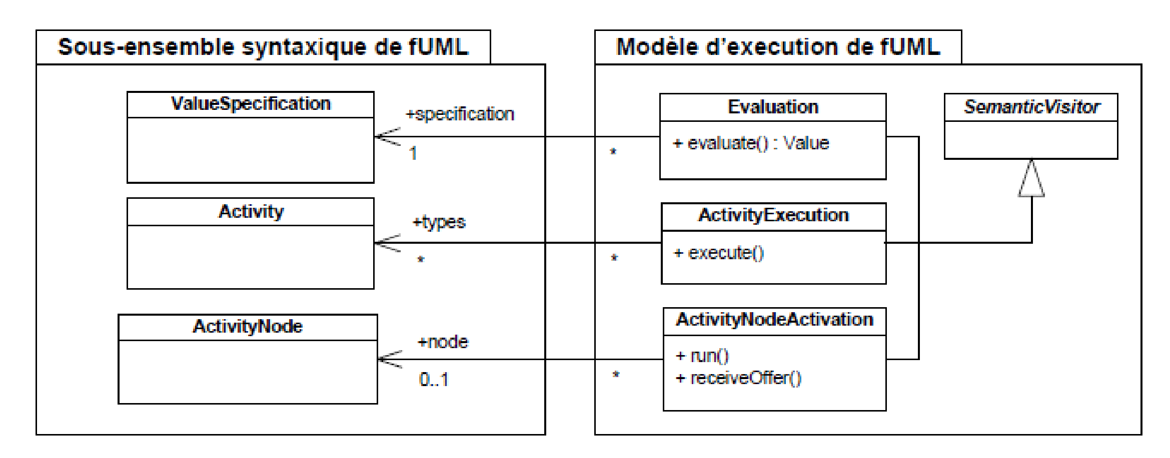


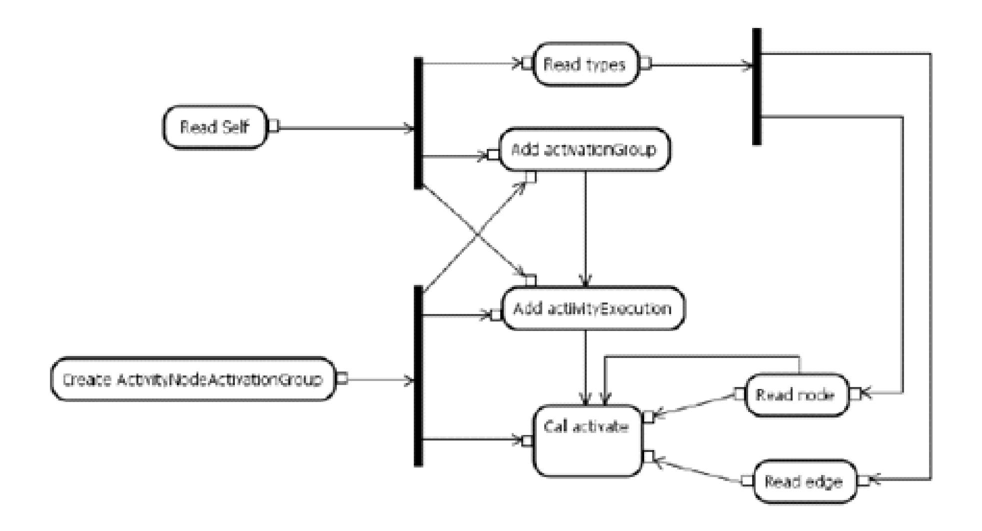
Figure 19: Le modèle d'exécution de fUML montrant les types de classes visiteurs

- **Les classes visiteurs de type Execution :** où le comportement de chaque sous classe de **Behavior** de la syntaxe abstraite inclus dans fUML est décrit par ce type de visiteur. Par exemple la classe visiteur **ActivityExecution** décrit le comportement des activités. Nous trouvons aussi les deux classes visiteurs **OpaqueBehaviorExecution** et **FunctionBehaviorExecution** qui décrivent respectivement **OpaqueBehavior** et **FunctionBehvaior**.
- **Les classes visiteurs de type Activation :** ce type de classe décrit la sémantique des différents nœuds d'activité. Nous trouvons notamment la classe visiteur SendSignalActionActivation correspond à l'action SendSignalAction.
- **Les classes visiteurs de type Evaluation :** ce type de classe visiteur décrit l'évaluation de la sous classe de ValueSpecification. Notons la classe visiteur du modèle d'exécution LiteralBooleanEvaluation spécifie comment une valeur booléenne (instance de la classe LiteralBoolean) est évaluée.

Chaque classe visiteur du modèle d'exécution comprend des opérations qui ont pour rôle de capturer la sémantique d'exécution. Le comportement des opérations est décrit sous la forme d'activités fUML. Ces activités sont spécifiées dans le langage Java,   ce dernier est utilisé comme une syntaxe concrète en remplaçant les diagrammes d'activités.

La figure 20 présente un exemple dont la partie supérieure montre un diagramme d'activitépartiel de l'opération **execute()** de la classe

ActivityExecution du modèle d'exécution. Et la partie inférieure de la figure 10 montre sa représentation équivalente en Java dans la condition de respecter le passage de Java vers les activités définit dans fUML.




```
Activity activity = (fUML.Syntax.Activity) (this.types.getValue(0));
ActivityNodeActivationGroup group = new ActivityNodeActivationGroup () ;
this.activationGroup = group;
group.activityExecution = this;
```

Figure 20: Un diagramme d'activité partiel d'une opération du modèle d'exécution et sa description en java

2.3.1. *Points de variation sémantique*

Le traitement des points de variation de fUML est effectué en utilisant la classe SemanticStrategy dans le modèle d'exécution [44], pour chaque point de variation sémantique ayant une opération abstraction, on définit une classe abstraite stratégie. En outre afin d'implémenter plusieurs comportements, d'où la nécessité de définir des sous classes concrètes où chacune de ces classes correspond à un point de variation en définissant une variante sémantique.

Dans le modèle d'exécution de fUML, deux points de variations sont pris :

Lors de la réception des évènements par un objet, Le premier point effectue un choix entre les évènements reçus, c'est-à-dire. Comment les évènements et dans quel ordre vont être traités.

GetNextEventStrategy est la classe stratégie abstraite et la classe concrète FIFOGetNextEventStrategy qui implémente une politique FIFO afin de traies les évènements reçus par ordre d'arrivée.

Le deuxième point de variation sémantique détermine la méthode qu'on utilise pour l'appel polymorphique d'une opération. Chaque opération a un

ordre dans une hiérarchie des classes et une opération peut avoir plusieurs comportements selon son ordre dans la hiérarchie. Nous trouvons par exemple l'opération dispatch de la classe Object définie la sémantique de la détermination de l'opération à appeler. Cette opération correspond à la classe abstraite

La classe stratégie abstraite correspondante est DispatchStrategy. La sémantique par défaut dans le modèle d'exécution est définie par la classe concrète RedefinitionBasedDispatchStrategy.

Les deux points de variation du modèle d'exécution dans fUML sont présentés dans la figure

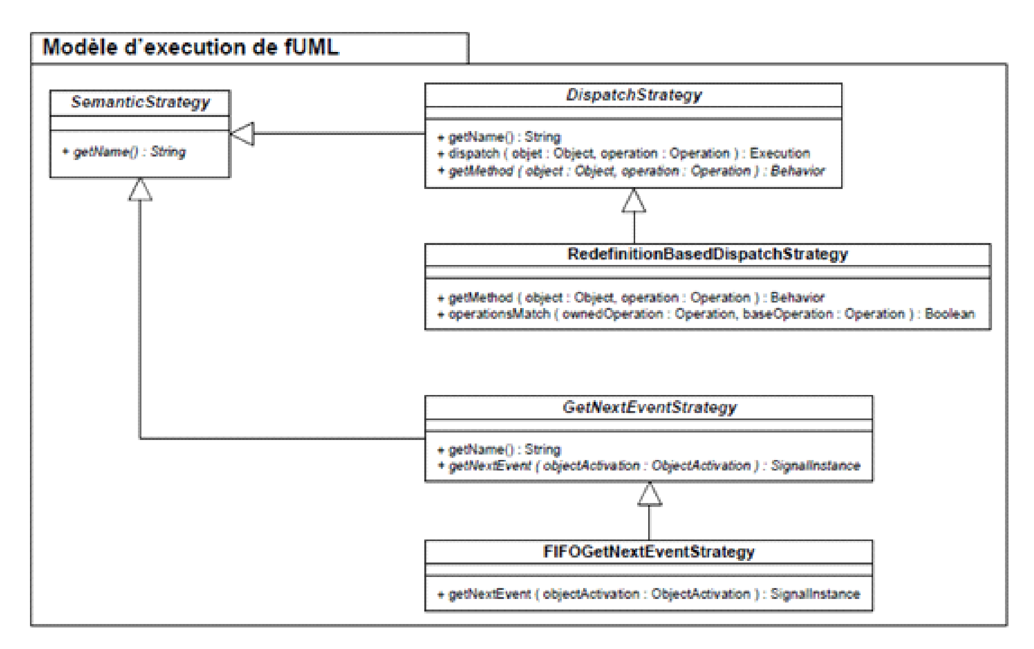


Figure 21:Les deux points de variation dans le modèle d'exécution dans fUML

2.4. Le moteur d'exécution et son environnement

Le moteur d'exécution et de fUML et son environnement correspond au package Loci présenté dans la figure 22.

Le package Loci comprend trois classes : Locus, Executor et ExecutionFactory. Ces trois classes correspondent à une machine virtuelle ou un interpréteur de modèle fUML.

- La classe **Locus** joue le rôle d'une unité de calcul ou un processeur. Elle est responsable à l'exécution des modèles fUML. Tous les objets crées pendant l'exécution sont placés dans cette classe.
- La classe **Executor** représente une interface avec laquelle nous pouvons accéder au moteur d'exécution. Les services nécessaires pour le déclenchement des exécutions des modèles sont donnés par cette classe.

- La classe **ExecutionFactory** crée les instances de toutes les classes visiteurs du modèle d'exécution de fUML.

Pour pouvoir lancer une exécution, la première étape est de définir un environnement d'exécution initial contenant les objets suivants :

- Une seule instance de la classe Locus.
- Une seule instance de la classe Executor.
- Une seule instance de la classe ExecutionFacotry.
- Une seule instance par chaque sous-classe concrète de type stratégie.

Afin de déclencher une exécution, un ensemble des étapes d'initialisation du moteur d'exécution doit être suit en commençant par la classe main:

- Instanciation des classes dans le but de créer et configurer l'environnement d'exécution.
- Appel de l'opération execute() de la classe ExecutionEnvironement en prenant comme paramètre l'activité à exécuter afin d'initialiser l'exécution. La même activité est prise aussi comme paramètre lors de l'appel de l'opération execute() de la classe Executor.

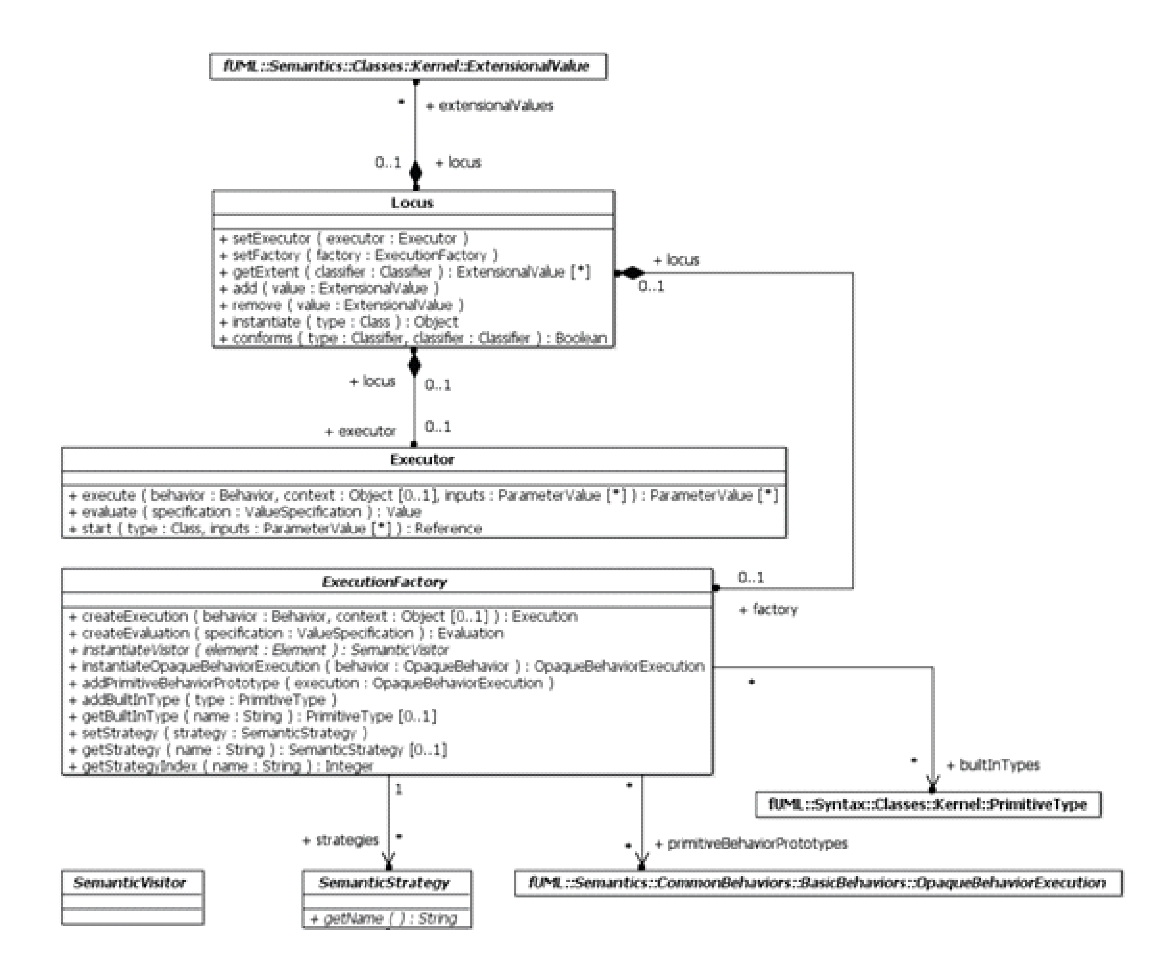


Figure 22: Le package Loci du modèle d'exécution de fUML

- Création de la classe visiteur qui correspond à l'activité prise comme paramètre lors de l'appel de l'opération createExecution() de la classe ExecutionFactory.

L'exécution d'une activité s'effectue par l'appel de l'opération execute() de la classe ActivityExecution.

Les étapes d'initialisation du moteur d'exécution sont présentées dans la figure 23 par un digramme de séquence.

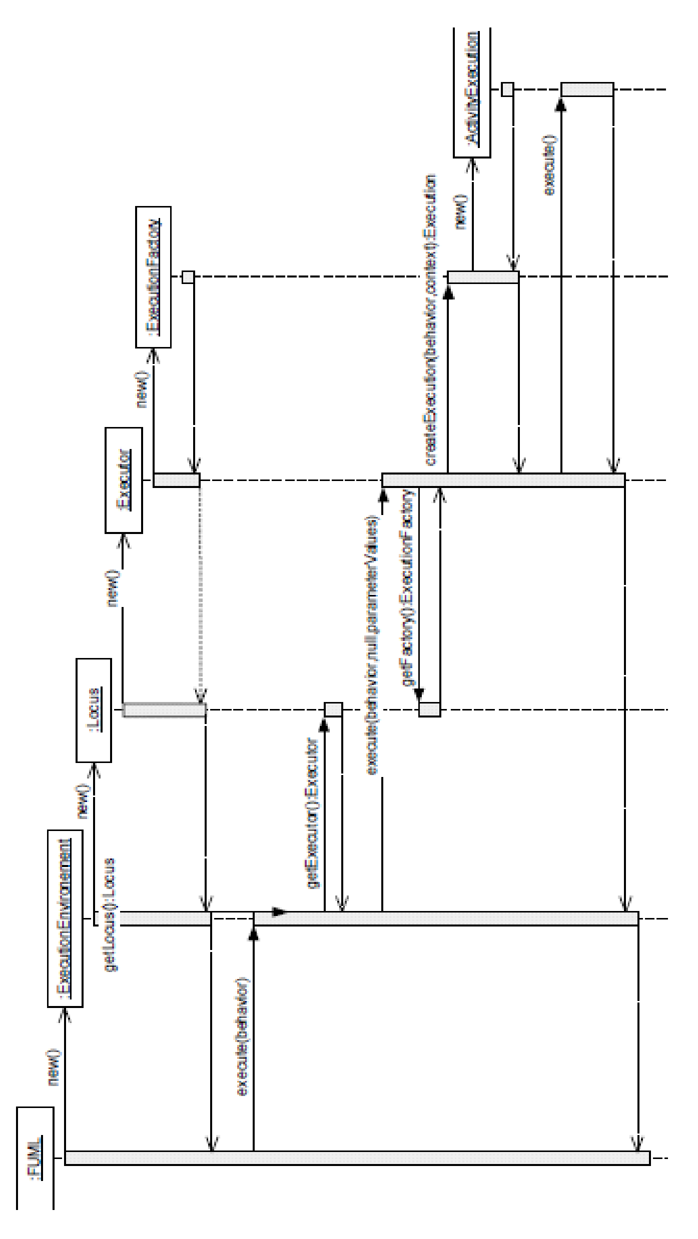


Figure 23: Diagramme de séquence de l'initialisation du moteur d'exécution

2.5. L'exécution des activités

Après l'initialisation du moteur d'exécution, l'exécution de l'activité est lancée en appelant l'opération execute() sur l'instance de la classe visiteur du modèle d'exécution. L'opération execute() crée toute les instances des classes visiteurs nécessaires pour accomplir l'exécution de l'activité. Ces instances correspondent aux nœuds compromis dans l'activité.

La figure 24 montre un diagramme d'activité d'une simple activité.

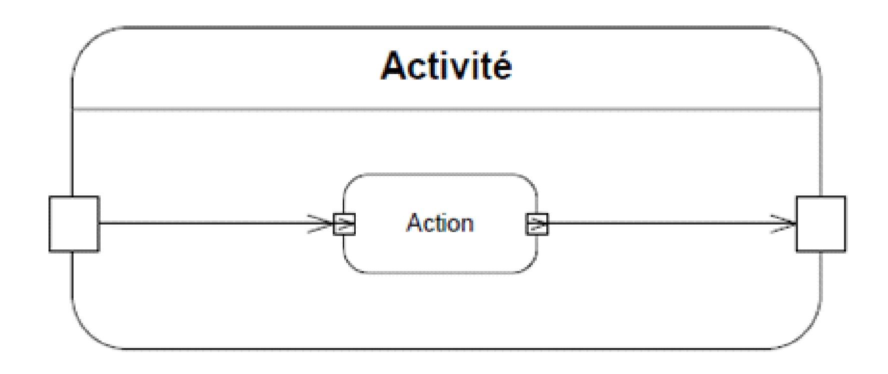


Figure 24: Un diagramme d'activité d'une simple activité

La figure 25 est partagée en deux parties : la partie gauche montre les éléments de la syntaxe abstraite du diagramme d'activité, et la partie droite montre les instances des classes visiteurs crées du modèle d'exécution.

Syntaxe abstraite | Modèle d'exécution

Figure 25: La syntaxe abstraite et le modèle d'exécution d'une simple activité

Durant l'exécution, plusieurs appels entre les instances des classes sont effectués, tout en respectant les étapes suivantes :

- Entrer les valeurs des paramètres d'entrée de l'activité à exécuter. Une vérification est réalisée par le moteur d'exécution afin de vérifier si toutes les valeurs nécessaires aux nœuds d'entrée sont fournies.
- Définir les nœuds activés dans l'activité, où l'exécution se lance à partir ces nœuds. Les nœuds activés sont les nœuds initiaux, les nœuds de paramètres d'entrées, et les actions n'ayant pas des arcs entrants.
- Envoyer un jeton à chaque nœud activé.
- Exécuter le comportement du nœud activé. Dans le cas ou plusieurs nœuds sont activés, le choix d'un nœud parmi eux est faite selon l'ordre de création dans le modèle d'exécution. Dans le cas contraire, le choix d'un nœud se fait en tant compte des dépendances de données et de contrôles entre les nœuds.

L'exécution du comportement d'un nœud suit les étapes suivantes :

- Vérifier si tous les jetons nécessaires pour l'exécution du nœud sélectionné sont présents, si cela est vérifié, on peut conclure que le nœud sélectionné est prêt à être exécuté.
- Vérifier si toutes les pré-conditions sont disponibles, si cette condition est vérifiée le nœud consomme ses jetons, sinon un autre nœud sera sélectionné, et on reviendra à l'étape 1.
- Lancer l'exécution du comportement du nœud. Des jetons seront produits comme des résultats de cette exécution.
- Emettre les jetons de sortie aux nœuds successeurs.
- Exécuter le nœud sélectionné en répétant les étapes de 1 à 4.
- L'exécution de l'activité termine lorsque tous les nœuds de cette activité sont exécutés. Et le résultat final de l'exécution de l'activité sera placé dans les nœuds de paramètre de sorties.

Chapitre 3 Contribution

Nous avons montré dans le chapitre précédent que toute exécution suivant le modèle d'exécution de fUML est effectuée sans l'intervention du concepteur.

Nous avons souligné ainsi l'absence d'une entité explicite responsable de partitionnement du système et la simulation d'exécution. La résolution de ces deux problèmes nécessite des extensions du modèle d'exécution de fUML.

Notre idée repose sur deux points :

Dans un premier temps, nous proposons de rompre l'exécution des appels d'opération en donnant la main à une entité qui décidera en fonction d'une politique de partitionnement proposée par le concepteur, quelle est la prochaine action à exécuter tout en respectant les choix du concepteur en ce qui concerne :

1. Choisir une implémentation matérielle ou logicielle de chaque fonctionnalité ; pour faire le partitionnement.
2. Définir pour chaque fonctionnalité son composant sur lequel va être implémentée.

Dans un second temps, nous introduisons un simulateur dans le modèle d'exécution afin de simuler les exécutions des différentes actions, nous avons besoin de :

- Le temps d'exécution de chaque fonctionnalité et sa date au plutôt et au plutard et définir l'ordonnancement des fonctionnalités.

La figure 26 montre d'une façon générale le travail effectué.

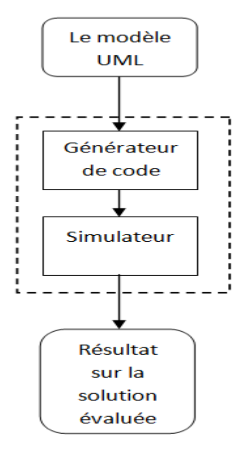


Figure 26: Vue générale sur la chaine d'outils proposée

1. Présentation du système

1.1. Diagrammes de cas d'utilisation :

L'outil proposé contient six cas d'utilisation :

1. Introduire le modèle UML
2. génération de code du modèle exécutable fUML
3. Simuler la solution

La figure 27 suivante montre le diagramme de cas d'utilisation de notre outil :

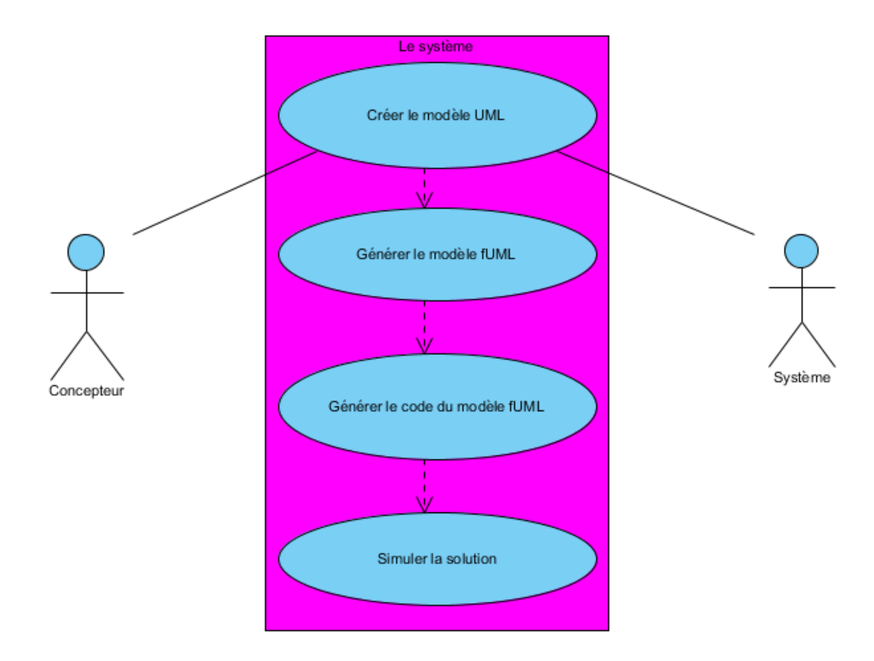


Figure 27: Cas d'utilisation de la chaine d'outils

1. *<u>Le 1er cas d'utilisation : Introduire le modèle UML du système conçu :</u>* le concepteur introduit le modèle UML proposé du système conçu.

   Le diagramme de classes est le diagramme utilisé dans fUML.

2. *<u>Le 2ème cas d'utilisation : génération de code du modèle exécutable fUML:</u>* Dans cette étape, le système génère le modèle d'exécution fUML en intégrant les trois tâches : le partitionnement, l'ordonnancement et l'allocation des ressources selon le choix effectué par le concepteur afin d'avoir un système conçu.

   Ce cas d'utilisation dépend du cas d'utilisations précédent « introduire le modèle UML », en plus il a besoin d'autres informations complémentaires en ce qui concerne : le partitionnement et l'ordonnancement. Dans ce cas, le système proposé fait appel au sous-système de dialogue, ce dernier interagit avec le concepteur, afin de lui demander de saisir toutes les informations nécessaires dans la solution proposée c'est-à-dire que le concepteur définie toutes les informations

supplémentaires qui n'ont pas introduits dans le modèles UML, ces informations dépendent de la solution proposée afin d'accomplir cette dernière. Pour chaque fonctionnalité (classe ou activité), le concepteur doit fournir les informations suivantes :

- Le placement de la tâche : Hard ou Soft ; le concepteur définie le type de la réalisation si elle est matérielle ou logicielle afin d'atteindre le partitionnement des fonctionnalités.
- le concepteur défini sur quel composant la fonctionnalité est implémentée, autrement dit la ressource allouée pour chaque fonctionnalité.
- La date d'exécution de la tâche (son ordonnancement).

3. *Le 3ème cas d'utilisation : Evaluer la solution :* C'est l'étape qui va emmètre le résultat au concepteur, son rôle est de valider et renseigner le concepteur sur la qualité de solutions trouvées, la solution proposée. on utilise des estimateurs (estimations des performances, des coûts, ...) afin de prédire les résultats de la conception avant et sans aller jusqu'à la réalisation du système.

   Dans ce cas, le système proposé fait appel une autre fois au sous-système de dialogue, ce dernier réintégrait avec le concepteur, afin de lui demander de saisir toutes les informations nécessaires dont il a besoin. Le concepteur définie toutes les informations supplémentaires qui n'ont pas introduits dans le modèles UML. Pour chaque fonctionnalité (classe ou activité), le concepteur doit fournir les informations suivantes :

   - Le temps d'exécution de chaque activité,
   - La date au plutôt et la date au plutard d'exécution de la fonctionnalité.

   Cette solution est évaluée ; Si après validation, le système considère que le choix de concepteur a donné un bon partitionnement (allocation des ressources et ordonnancement) autrement dit la qualité de la solution est satisfaisante, le concepteur passe à l'étape suivante de la conception. Sinon, il refait/revoie son partitionnement et réexecute à nouveau notre outil pour valider le partitionnement de nouveau.

2. La préparation de l'environnement de travail

1. installer java 6 ;
2. Installer "Eclipse-modelling" c'est ce package qui comporte tous les éléments nécessaire à la modélisation (uml, emf, gmf, etc ...);
3. A partir de la plateforme Eclipse, installer "Acceleo" et "papyrus" à partir du menu "Help/Install modeling Components" ;

4. Pour le projet fuml, il suffit de dézipper la source et importer le projet.

3. La réalisation du système

Le concepteur lance le fichier exécutable **system.exe,** qui provoque le déclanchement du système d'évaluation de la solution, une fenêtre s'affiche avec lequel il va interagir, la figure 27 montre l'interface du système conçu :

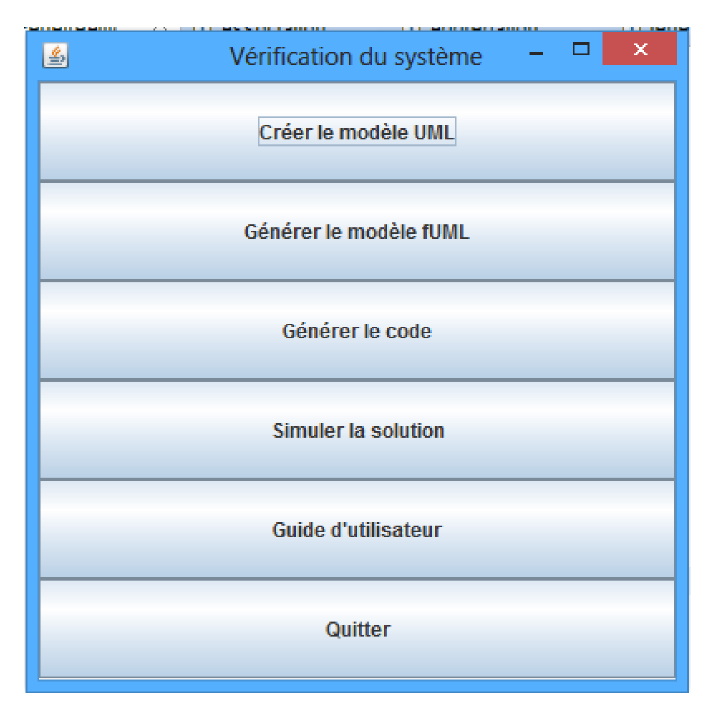


Figure 28: Interface du système conçu

Sur cette interface, nous trouvons six fonctions :

2.1 *Définir le modèle UML :* lorsque le concepteur clique sur le bouton **« Créer le modèle UML »**, une zone de dessin s'affiche avec les outils nécessaires de réalisation d'un modèle UML, ou le concepteur crée son modèle UML proposé de son système à concevoir.

Pour créer la structure d'un modèle les diagrammes de classes UML habituelles doivent être utilisés.

L'éditeur de diagramme de classes est intégré à l'éditeur du langage d'action textuel, qui est utilisé pour créer le comportement de chaque opération. L'intégration se réfère à la capacité de choisir une classe ou une opération à partir du modèle et, avec une action simple (double-clique ou une touche raccourci), l'éditeur de langage d'action pour les comportements peut être ouvert et utilisé pour la sélection des éléments. L'éditeur du langage d'action doit être capable de charger le comportement existant sous la sélection d'élément et l'afficher correctement. En outre, l'action de sauvegarde, il faut bien mettre à jour le modèle sous l'élément sélectionné.

2.2 *Générer le modèle fUML :* cette fonction est provoqué par le clic sur le bouton **« Générer le modèle fUML »** ; le rôle de cette fonction est la création du modèle exécutable fUML.

2.3 *Effectuer une génération de code du modèle exécutable fUML:* cette fonction déclenche l'outil de génération de code. Cette fonction ne peut pas s'exécuter sans que le concepteur à déterminé le modèle UML avec toutes les extensions manquantes en tenant compte des contraintes de partitionnement HW/SW et avoir un comportement du système partitionné selon le choix effectué par le concepteur. En outre lors de lancement de cette fonction, une autre fenêtre s'affiche afin d'insérer toutes les informations complémentaires que le concepteur doit fournir sur le modèle UML, qui ne sont pas inclus dans ce dernier, et qui sont nécessaires

Pour chaque fonctionnalité (classe ou activité), le concepteur détermine :

- Son ordonnancement (sa date d'exécution) ;
- Le type de la réalisation : soit matérielle ou logicielle.
- La ressource de l'architecture à allouer pour chaque activité

Après l'achèvement de cette fonction, nous obtenons un code généré.

2.4 *Evaluer la solution :* Cette fonction joue le rôle d'un simulateur afin d'analyser/étudier le comportement d'un système embarqué. Cette fonction ne peut pas être exécutée sans que la génération de code du système soit effectuée (fonction 05). Notre outil étudie la solution. Le résultat de cette fonction est l'estimation de la qualité de la solution ou l'implémentation proposé par le concepteur (estimation de performances, estimation de coût,…etc).

2.5 *Guide d'utilisation :* Cette fonction est lancée en cliquant sur le bouton **« guide d'utilisation »** ; elle fournie un manuel d'utilisation afin de permettre le concepteur à comprendre l'application et la fonction de chaque bouton.

2.6 *Quitter :* le concepteur peut quitter l'application en cliquant sur le bouton **« Quitter ».**

4. Vue générale sur l'outil proposé

L'ordonnancement de fUML, le partitionnement et l'allocation des ressources : ces trois problèmes sont indépendants mais l'idéal est de les réaliser simultanément.

Le problème de partitionnement inclut le problème d'ordonnancement et l'allocation des ressources ; parce que le choix de la ressource allouée est relié avec les dépendances des fonctionnalités (ordonnancement).

Le but de ce travail est de mettre en place une base d'outils comportant un générateur de code et un simulateur afin d'analyser/étudier le comportement d'un système embarqué très tôt (dès la phase de spécifications).

Le modèle exécutable fUML ne prend pas en considération ces trois aspects, pour soulever ces problèmes nous avons fait une extension d'une entité explicite dans le modèle fUML responsable à étudier le comportement du système, dans le but est de renseigner le concepteur sur la qualité de la solution trouvée selon son choix de réalisation des fonctionnalités.

Pour cela nous avons proposé une base d'outils générateur de code et un simulateur) afin d'étudier et analyser, qui composé de deux entités importantes :

A. L'entité Interface ;

B. L'entité Générateur de code ;

C. L'entité simulateur.

4.1. L'entité Interface

Cette entité va être comme une interface entre le concepteur et le système. Cette entité est responsable à l'interaction et la réception des données fournies par le concepteur :

4.1.1. Le modèle UML

Lorsque le concepteur crée son modèle UML, cette entité sauvegarde cette modélisation.

4.1.2. Les informations supplémentaires ajoutées par le concepteur :

Le modèle UML crée par le concepteur n'est pas complet d'un point de vue informations sur les activités concernant l'architecture nécessaires à évaluer le partitionnement proposé par le concepteur.

Dans le cadre de notre projet nous supposons que le partitionnement a été réalisé par le concepteur en utilisant des moyens tiers (manuelle, outils), ces informations concernant le partitionnement sont utilisés dans la génération de code du système, donc nous avons besoin d'autre détails complémentaires, citons :

- Caractériser les composants une fois placé :
  - Le temps d'exécution ;
  - La date au plutôt et la date au plutard d'exécution ;
- Distinguer le placement hard/soft : Le type de la réalisation ;
- Définir l'ordonnancement ;
- Identifier le composant de l'architecture (Identifier CPU, circuit HW, etc) : La ressource sur la quelle la fonctionnalité va être implémentée.

5. Implémentation

5.1. Générateur de code

L'exécution est modifiée en ajoutant le modèle responsable du partitionnement, c'est à dire que les activités du modèle d'exécution du modèle fUML seront exécutées suivant la politique choisie par le concepteur. Cela veut dire que le choix d'une activité à exécuter dans le moteur d'exécution est défini par l'entité appelée **Générateur**. Cette entité suit l'ordre des activités et leurs partitionnements proposés par le concepteur. Cela veut dire que l'entité **Générateur** a deux rôles essentiels :

- Le partitionnement des activités (le type de partitionnement, sur quel composant va être implémenté, L'ordonnancement des activités exécutées et qui sont exécutées par le même composant),
- Le choix de l'action à exécuter

Pour effectuer le partitionnement, l'entité **Générateur** utilise les informations introduites par le concepteur en ce qui concerne le partitionnement, c'est-à-dire elle suit le choix de partitionnement proposé par le concepteur.

L'entité GENERATEUR est présentée par la classe GENERATEUR. Il manipule une liste des activités. Il a comme entrées la liste des ActivityNodeActivation, qui sont les actions prêtes à être exécutées.

La classe **Générateur** contient les opérations suivantes :

- Le comportement de la classe **Générateur** dans le moteur d'exécution de fUML est déclenché.
- Le partitionnement des activités. Où elle suit le choix effectué par le concepteur selon sa solution proposée. Autrement dit c'est le concepteur qui précise l'ordre de chaque activité, sa ressource de l'architecture ainsi que son type d'implémentation.
- Après le partitionnement de toutes les activités, une autre opération est déclenchée qui est responsable à la sélection d'une activité parmi la liste, cette activité est envoyée au locus afin d'être exécutée.
- Une instruction de mise à jour la des activités de la classe **Générateur** en retirant l'action exécutée.

Pour donner la main au générateur pour définir quelle action va être exécutée, son comportement est ajouté dans le moteur d'exécution en modifiant les appels d'exécution des actions.

Le comportement de générateur est déclenché.

- Partitionnement : cette instruction fait appel à l'algorithme de partitionnement, afin d'avoir à la fin un système partitionné.
- **ExécuterAction** : cette opération lance la fonction **exécuter()** pour l'exécution la prochaine action à exécuter.
- **MettreAJour** : la liste de générateur est mise à jour.
- L'opération ExecuterAction est appelée une autre fois pour exécuter la prochaine action.

Nous avons signalé que la politique de partitionnement et d'ordonnancement est une politique choisie par le concepteur.

Cette politique est présentée par un algorithme qui est en faite une interface utilisateur qui interagit avec le concepteur afin de fournir les informations nécessaires, en ce qui concerne le partitionnement et l'ordonnancement :

- Sur quel composant, une fonctionnalité va être implémentée,
- Quel type d'implémentation de chaque fonctionnalité,
- Son ordonnancement (sa date d'exécution)

Cette interface est présentée dans la figure suivante :

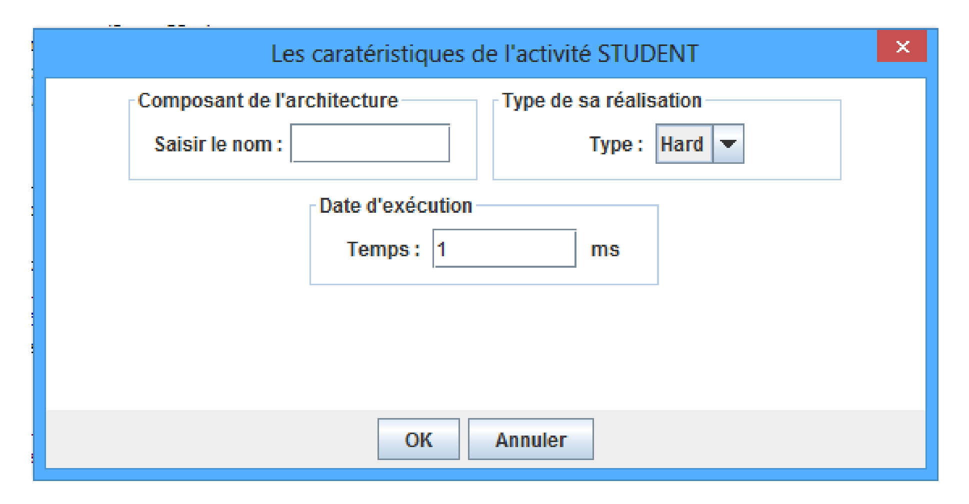


Figure 29: Interface de concepteur pour e partitionnement

Cet algorithme d'ordonnancement et partitionnement est présenté par la classe : **PolitiqueParitionnementANDOrdonnancement**.

Une fois le générateur est démarré, il donne la main au concepteur pour partitionner son système.

5.2. Simulation

Après génération du modèle exécutable (fUML) du système embarqué, ce code qui a tenu en compte les contraintes de partitionnement HW/SW. Nous venons à l'étape de simulation, où nous faisons une vérification des propriétés attendues.

Pour cela nous avons besoin les informations suivantes :

- Le temps d'exécution de chaque activité,
- Sa date au plutôt et plutard d'exécution.

L'entité **Simulateur** introduit. Cette entité est présentée par la classe **Simulateur**, son rôle est de simuler le comportement du code généré du modèle d'exécution fUML.

Ce simulateur va s'intéresser du temps d'exécution des activités, si les activités sont exécutées dans leur intervalle du temps (date au plutôt, date au plutard), calculer sa performance du système.

Les informations utilisées par ce simulateur, qui sont citées au dessus sont ajoutées par le concepteur lors de déclenchement de la simulation. Cela est effectué par un algorithme qui est en faite une interface concepteur qui interagit avec le concepteur afin de fournir les informations nécessaires.

Cette interface est présentée dans la figure suivante :

les caractéristiques du temps de l'activité InsertEtudiant
Temps d'exécution
Temps : 10 ms
Date au plutot
Date : 10 ms
Date au plutard
10 Date :
OK
Annuler

Figure 30: Interface de concepteur pour l'ajout des informations concernant le temps d'exécution

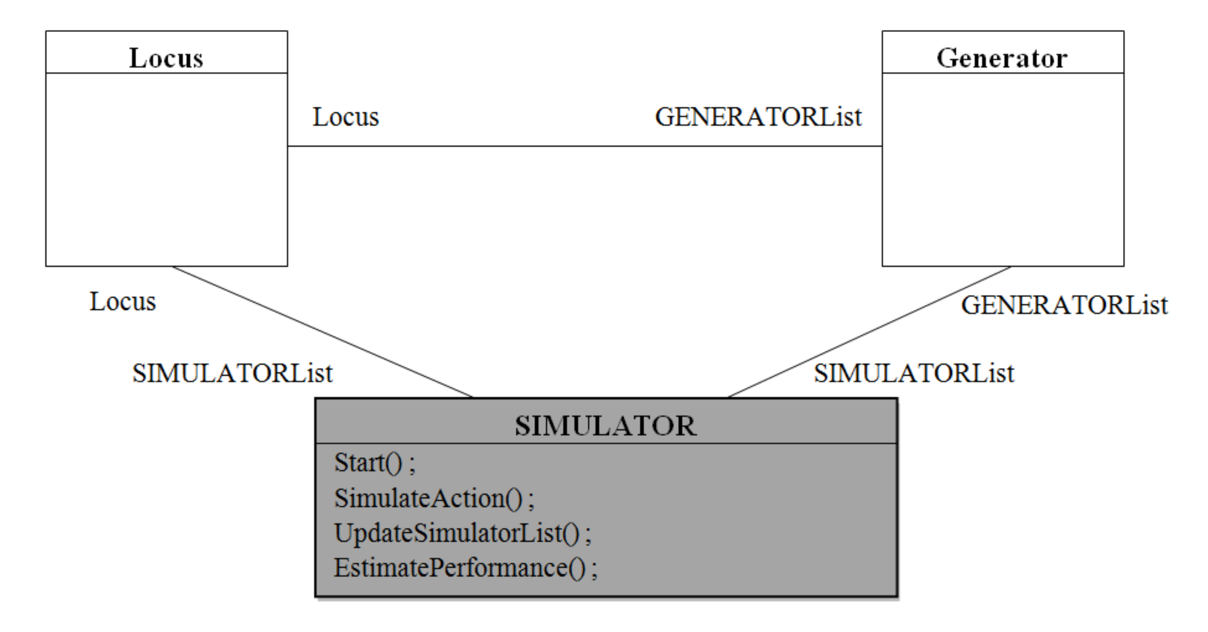


Figure 31: Description du simulateur introduit dans le modèle d'exécution fUML

Le simulateur interagit avec le locus, afin de simuler l'exécution de chaque activité.

Le comportement de simulateur contient les opérations suivantes :

- Le comportement du simulateur est démarré par l'opération Start,
- Lors de le déclanchement de l'opération **ExecuterAction** est déclenchée, l'opération **SimulerAction** est déclenchée afin de récupérer les données suivantes de chaque activité :
  1. Est-ce que l'opération est exécutée dans l'intervalle défini par le concepteur (date au plutôt, date au plutard),
  2. Le temps d'exécution de l'activité est le temps d'exécution défini par le concepteur,
     - L'opération **MettreAjourListeGénérateur** est déclenchée afin de retirer l'activité simulée.
     - **EstimerPerformance** est une opération qui déclenche l'exécution d'une formule de calcul de performance afin de rendre au concepteur la qualité de sa solution.

Remarque : Toutes les informations ajoutée précédemment par le concepteur, revient à l'ajout d'une nouvelle information au niveau syntaxique, et définir sa sémantique.

6. Conclusion

Nous avons résolu le problème de partitionnement dans fUML, en insérant un générateur dans le modèle d'exécution de fUML, son rôle est fournir un système où toutes les fonctionnalités sont partitionnées.

Après avoir un système partitionné, nous avons simulé son comportement afin d'avoir une estimation sur la qualité du système partitionné.

Résumé

Dans ce projet, nous traitons le problème de vérification des systèmes embarqués, où l'idée est de mettre en œuvre une chaine d'outils comportant un générateur de code et un simulateur pour pouvoir évaluer la solution proposée le plutôt possible dans le flot de conception et avant de passer à la phase de réalisation.

Pour cela, nous avons exploité le standard fUML (Fonadationnal for subset UMLà offert par l'OMG. Comme son nom l'indique, ce standard présente la sémantique d'exécution d'un sous ensemble d'UML.

Le concepteur modélise la structure et le comportement de son système, ensuite ce modèle est exécuté par ce standard afin de vérifier le système conçu.

Dans cette thèse, nous avons présenté les principaux concepts de l'ingénierie dirigée par les modèles IDM, ainsi que une vue détaillée sur les systèmes embarqués. Ensuite dans le chapitre de l'état de l'art, nous avons présenté les méthodes de vérifications existantes, après nous avons détaillé le standard fUML, et ses aspects en identifiant ses limites selon les besoins des systèmes embarqués, où les différents réalisations sont :

- Nous avons introduit un partitionneur, qui a pour rôle de partitionner les activités, pour chacune de ces dernières nous avons spécifié son type d'implémentation, sur quel composant va être exécutée, son ordre d'ordonnancement.
- Nous avons introduit un simulateur, il prend en compte le temps d'exécution de chaque activité, sa date au plutôt, sa date au plutard. Ce simulateur calcule le temps d'exécution du système, estimer sa performance, et voir si le système répond aux spécifications exigées.

Afin d'effectuer ce travail, nous avons proposé les extensions nécessaires, qui sont implémentée sous forme d'un plugin intégré dans fUML.

Perspectives

Les perspectives des travaux précédents peuvent avoir plusieurs chemins :

- Nous devons enrichir la sémantique d'exécution du sous ensemble d'UML par l'ajout de d'autres diagrammes.
- Nous devons proposer un plugin pour que les traces soient visualisées.

[1] Audrey MARCHAND ; Cours de l'UML pour le temps réel et l'embarqué ; Module Me1 Ingénierie de logiciel ; 2005-2006.

[2] TÖRNGREN M., Fundamentals of implementing real-time control applications in distributed computer systems, Journal of Real-Time Systems, 14, 219-250, 1998.

[3] Bran Selic. Using uml for modeling complex real-time systems. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, LCTES '98, pages 250–260, London, UK, 1998. Springer-Verlag.

[4] Bran Selic. Using uml for modeling complex real-time systems. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, LCTES ’98, pages 250–260, London, UK, 1998. Springer-Verlag.

[5] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. UPGRADE – The European Journal for the Informatics Professional, 5(2) :21–24, 2004.

[6] Jean Bézivin. La transformation de mod(é)les. In Ecole d’Eté d’Informatique CEA EDF INRIA, cours no 6, 2003.

[7] E. Seidewitz. What models mean. Software, IEEE, 20(5) :26 – 32, sept.-oct. 2003.

[8] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In ASE, 2001.

[9] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci., 152 :125–142, March 2006.

[10] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci., 152 :125–142, March 2006.

[11] Jean marie Favre. Towards a basic theory to model model driven engineering. In In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004, 2004.

[12] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. UPGRADE – The European Journal for the Informatics Professional, 5(2) :21–24, 2004.

[13] Thomas Stahl and Markus Völter. Model Driven Software Development : Technology, Engineering, Management. John Wiley & Sons, 2006.

[14] Etienne Juliot et Stéfane Lacrampe Jonathan Musset. Acceleo 2.6 : Guide utilisateur. Obeo, April 2008.

[15] Thomas Stahl and Markus Völter. Model Driven Software Development : Technology, Engineering, Management. John Wiley & Sons, 2006.

[16] Object Management Group, Inc. Meta Object Facility (MOF) Core Specification Version 2.4.1. Technical Report formal/2011-08-07, OMG, 2011.

[17] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. Technical Report formal/2011-01-01, OMG, 2011.

[18] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied Metamodelling - A Foundation for Language Driven Development. CETEVA, second edition, 2008.

[19] David Harel and Bernhard Rumpe. Meaningful modeling : What's the semantics of "semantics"? Computer, 37 :64–72, October 2004.

[20] Van Hon BUI ; Génération de code à partir d'un diagramme d'états –transitions ; mémoire en vue de l'obtention du Master d'ingénieur civil en informatique à finalité spécialisée en ingénierie informatique ; Faculté des sciences appliquées – Département IRIDIA CoDe – Université libre de BRUXELLES ; 2010.

[21] P.KADIONIK ; Cours de l'option Systèmes embarqués ; Ecole Normale Supérieure d'Electronique, Informatique et Radiocommunications de Bordeaux – ENSEIRB ; 2004.

[22] Mohamed KOUDIL ; Cours intitulé – Méthode de conception conjointe des systèmes embarqués ; Institut national de formation et informatique ; 2004.

[23] Eugenia Gabriela NUTA NICOLSCU ; Spécification et validation des systèmes hétérogènes embarqués ; Thèse pour obtenir le grade de DOCTEUR de l'INPG en microélectronique ; Laboratoire TIMA TIMA dans le cadre de l'Ecole Doctorale EEATS; Institut national polytechnique de GRENOBLE ; 2002.

[24] Lovic GAUTHIER ; Génération des systèmes d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques ; Thèse pour obtenir le grade de DOCTEUR de l'INPG en microélectronique ; Laboratoire TIMA dans le cadre de l'Ecole Doctorale EEATS ; Institut national polytechnique de GRENOBLE ; 2001.

[25] Frédéric ROUSSEAU ; conception des systèmes logiciel/matériel : du partitionnement logiciel/matériel au prototypage sur plateformes reconfigurables ; Thèse d'habilitation à diriger des recherches ; Université Joseph Fourier ; 2005.

[26] Jean-Paul JAMONT ; DIAMOND : Une approche pour la conception des systèmes multi-agents embarqués ; Thèse pour obtenir le grade de DOCTEUR de l'INPG en informatique ; Laboratoire de Conception et d'Ingénierie des systèmes ; Institut national polytechnique de GRENOBLE ; 2005.

[27] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. PAVIOT, S. YOO, A.A. Jerraya, M. Diaz-Nava, "Component-Based Design Approach for Multicore SoCs", DAC'02, June 10-14 2002, New Orleans, USA, 2002.

[28] Mohamed KOUDIL ; Cours de la Modélisation ; Ecole nationale supérieure d'informatique ; 2012.

[29] Michel ISRAEL et Denis DUPONT ; De la spécification formelle au partitionnement matériel/logiciel ; Laboratoire de mathématique et d'informatique ; 1997.

[30] D. LUCKMAN et al., “Specification and Analysis of system architecture using RAPIDE”, IEEE on software engineering – special issue on software architecture, 21(4), April, 1995.

[31] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In Proceeding of the 2nd International Symposium on Programming, Paris, France, 1976. Dunod.

[32] C. A. Petri. KommunikationmitAutomaten. PhD thesis, Schriften des Institutes fur InstrumentelleMathematik, Bonn, West Germany, 1962.

[33] S. Christensen, J. B. Jørgensen, and Kristensen L. M. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, Proceedings of the 3rdInternational Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS’97), volume 1217 of Lecture Notes in Computer Science, pages 209–223, Enschede, Nederlands, April 1997. Springer-Verlag.

[34] G. Berthelot, C. Johnen, and L. Petrucci. PAPETRI : Environment for the Analysis of Petri Nets. In Proceedings of the 2nd International Workshop in Computer Aided Verification (CAV’90), volume 531 of Lecture Notes in Computer Science, New Brunswick, NJ, United States, June 1990. Springer-Verlag.

[35] B. Grahlmann. The Reference Component of PEP. In E. Brinksma, editor, Proceedings of the 3rd International Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS’97), volume 1217 of Lecture Notesin Computer Science, Enschede, Nederlands, April 1997. Springer-Verlag.

[36] R. Alur and D. L. Dill. A theory of timed automata. Theoretical Computer Science, 1994.

[37] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Hybrid Systems III, Verification and Control, volume 1066 of LNCS, Springer-Verlag, 1996.

[38] P. Pettersson et K. G. Larsen. UPPAAL. Bulletin of the European Association for Theoretical Computer Science, 70 :40–44, 2000. http://www.uppaal.com

[39] F.Laroussinie and K.G. Larsen. CMC : A tool compositional model-checking of real-time systems. In IFIP Joint Int. Conf. Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE-PSTV 98), Novembre 1998.

[40] Thomas A. Henzinger. The theory of hybrid automata. In Proceedings, 11th Symposium on Logic in Computer Science (LICS ’96), IEEE Computer Society Press, 1996.

[41] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech : A model checker for hybrid systems. In Proceedings of the 9th International Conference Computer Aided

Verification, volume 1254 of Lecture Notes in Computer Science, Springer, 1997.

[42] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[43] Object Management Group, Inc. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. Technical Report formal/2011-02-01, OMG, 2011.

[44] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.